ТРУДЫ

ИНСТИТУТА СИСТЕМНОГО ПРОГРАММИРОВАНИЯ РАН

PROCEEDINGS OF THE INSTITUTE FOR SYSTEM PROGRAMMING OF THE RAS

ISSN Print 2079-8156
Том 38 Дополнительный выпуск 1
Препринт 33
ISSN Online 2220-6426
Volume 38 Additional Issue 1
Preprint 33

Институт системного
программирования
им. В.П. Иванникова РАН

Москва, 2026

ИСП РАН

ISSN 2079-8156 (Print), ISSN 2220-6426 (Online)

Труды Института системного программирования РАН
Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областям системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge. **Proceedings of ISP RAS** is abstracted and/or indexed in:

УДК004.45

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.
Телефон: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Сайт: http://www.ispras.ru/proceedings/

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Tel: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Web: http://www.ispras.ru/en/proceedings

DOI: 10.15514/ISPRAS-2026-38-PREPRINT-33

Динамическое обнаружение зависимости от порядка вычислений в программах на Си

[1,2]А.Ю. Климов, ORCID: 0009-0004-3719-5873, <klimov.aiu@ispras.ru>
[2]А.В. Монаков, ORCID: 0000-0001-5118-0054, <amonakov@ispras.ru>
[2]В.А. Иванишин, ORCID: 0000-0002-9784-2911, <vlad@ispras.ru>
[2]Д.М. Мельник, ORCID: 0000-0002-0177-1558, <dm@ispras.ru>
[1]Московский физико-технический институт,
141701, Россия, Московская область, г. Долгопрудный, Институтский переулок, д. 9.
[2]Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В работе предложена модель обнаружения зависимости от порядка вычислений подвыражений в программах на языке Си. Модель представляет отношение sequenced-before стандарта C11 в виде иерархической структуры, допускающей эффективный поиск неупорядоченных и неопределённо-упорядоченных модификаций во время исполнения. Программная реализация модели в виде санитайзера для GCC произвела срабатывания на 139 из 4526 пакетов Alpine Linux 3.23, преимущественно фиксируя неопределённо-упорядоченные вычисления.

Ключевые слова: санитайзер; неопределённое поведение; порядок вычислений; GCC; компиляторная инструментация; динамический анализ

Для цитирования: Климов А.Ю., Монаков А.В., Иванишин В.А., Мельник Д.М. Динамическое обнаружение зависимости от порядка вычислений в программах на Си. Труды ИСП РАН, том 38, доп.вып. 1, Препринт ИСП РАН 33, 2026 г., 50 с. DOI: 10.15514/ISPRAS–2026–38-PREPRINT-33

Dynamic detection of evaluation-order dependence in C programs

[1,2]A.Yu. Klimov, ORCID: 0009-0004-3719-5873, <klimov.aiu@ispras.ru>
[2]A.V. Monakov, ORCID: 0000-0001-5118-0054, <amonakov@ispras.ru>
[2]V.A. Ivanishin, ORCID: 0000-0002-9784-2911, <vlad@ispras.ru>
[2]D.M. Melnik, ORCID: 0000-0002-0177-1558, <dm@ispras.ru>
[1]Moscow Institute of Physics and Technology,
9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.
[2]Ivannikov Institute for System Programming of the RAS,
25 Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. This paper proposes a model for detecting dependence on the evaluation order of subexpressions in C programs. The model represents the C11 sequenced-before relation as a hierarchical structure that enables efficient dynamic search for unsequenced and indeterminately-sequenced modifications at run time. A software implementation of the model as a sanitizer for GCC fired on 139 of 4526 Alpine Linux 3.23 packages, mostly flagging indeterminately-sequenced evaluations.

Keywords: sanitizer; undefined behavior; evaluation order; GCC; compiler instrumentation; dyna mic analysis

For citation: Klimov A.Yu., Monakov A.V., Ivanishin V.A., Melnik D.M. Dynamic detection of evaluation-order dependence in C programs. Trudy ISP RAN/Proc. ISP RAS, vol. 38, add. issue 1, Preprint ISP RAS 33, 2026. 50 p. (in Russian). DOI: 10.15514/ISPRAS–2026–38-PREPRINT-33

1. Введение

Язык программирования Си широко используется для разработки системного программного обеспечения: ядер операционных систем, встраиваемых систем, компиляторов и интерпретаторов. Являясь столь распространённым, он предоставляет программисту минимум защиты от ошибок. Ряд операций, с точки зрения стандарта, приводит к *неопределённому поведению* (undefined behavior, UB): переполнение знаковых целых, разыменование нулевого указателя, выход за границы массива и многие другие. Формально, программа, содержащая UB, не является корректной программой на Си.

Стандарт языка не даёт гарантий относительно поведения таких программ. Внешне они могут работать нормально, компилироваться без предупреждений и проходить все тесты. Однако при смене платформы, компилятора или входных данных программа, содержащая UB, может изменить своё поведение неожиданным образом. Известны и случаи, когда неопределённое поведение в коде открывало уязвимости. Так, в драйвере TUN/TAP ядра Linux указатель разыменовывался до проверки на NULL. Компилятор GCC, предположив, что разыменованный указатель не может быть нулевым, удалил проверку целиком. Это превратило ошибку в уязвимость CVE-2009-1897, позволявшую локальному пользователю получить привилегии суперпользователя [1].

Для обнаружения подобных ошибок существует широкий набор инструментов. Компиляторы GCC и Clang содержат встроенные статические анализаторы, выдающие предупреждения при компиляции. Внешние статические анализаторы, такие как Coverity, выполняют более глубокий анализ кода без его запуска. Динамические инструменты — Valgrind [2], AddressSanitizer [3], UndefinedBehaviorSanitizer [4] — обнаруживают ошибки во время выполнения программы: ошибки работы с памятью, целочисленное переполнение, разыменование нулевого указателя и другие.

Тем не менее, для некоторых видов неопределённого поведения существующие инструменты оказываются неэффективны. В настоящей работе рассматривается один из них — неупорядоченные модификации. В зависимости от контекста такие ситуации могут являться неопределённым поведением или приводить к тому, что результат исполнения определяется порядком вычислений, выбранным компилятором.

1.1 Порядок вычислений в языке Си

1.1.1 Побочные эффекты и порядок вычислений

Побочным эффектом (side effect) стандарт Си называет изменение состояния среды выполнения при вычислении выражения — модификацию объекта или файла, обращение к volatile-объекту, а также вызов функции, выполняющей любое из этих действий (C11, §5.1.2.3). Например, в выражении `i++` вычисление значения даёт текущее значение `i`, а побочный эффект увеличивает `i` на единицу. При этом порядок вычисления подвыражений в ряде контекстов стандарт не фиксирует (§6.5, параграф 3), оставляя его компилятору. Если два неупорядоченных вычисления обращаются к одному объекту и хотя бы одно из них выполняет запись, возникает неопределённое поведение.

Исторически стандарт C99 [6] выражал это правило через *точки следования* (sequence point) — точки исполнения, в которых все предшествующие побочные эффекты завершены, а последующие ещё не начаты: между двумя последовательными точками следования объект нельзя модифицировать более одного раза, а его прежнее значение читать разрешено лишь для вычисления нового. Нарушение — UB:

```
int i = 0;
printf("%d\n", i++ + i++);
```

Листинг 1. Неупорядоченные инкременты одной переменной

Здесь `i` модифицируется дважды между точками следования. Этот пример также демонстрирует смежность проблем неупорядоченных модификаций с состояниями гонки в многопоточном коде.

1.1.2 Модель упорядоченности (C11)

Стандарт C11 разделяет каждое вычисление на *вычисление значения* (value computation) и *побочный эффект* (side effect). Правило §6.5, параграфа 2 формулируется так: если побочный эффект над скалярным объектом не упорядочен (unsequenced) относительно другого побочного эффекта над тем же объектом или относительно вычисления значения, использующего этот объект — это неопределённое поведение.

Само понятие упорядоченности выражается через несколько инструментов, основным из которых является отношение частичного порядка *sequenced before* (§5.1.2.3, параграф 3), задающее порядок между парой отдельных вычислений. Другой, более общий инструмент — точка следования, или *sequence point*. Наличие точки следования между вычислениями A и B означает, что все побочные эффекты и вычисления значений, связанные с A, упорядочены (sequenced before) относительно всех побочных эффектов и вычислений значений, связанных с B. Далее в работе отношение sequenced-before изображается стрелкой $\xrightarrow{\text{S.B.}}$.

По умолчанию побочные эффекты и вычисления значений подвыражений не упорядочены (§6.5, параграф 3). Модель sequenced-before различает три вида отношений между вычислениями:

1. **Упорядоченные** (sequenced): одно вычисление завершается перед началом другого. Примеры: левый и правый операнды оператора-запятой, операнды `&&` и `||`.
2. **Неупорядоченные** (unsequenced): порядок не определён, и вычисления могут перекрываться. Если при этом два побочных эффекта обращаются к одному объекту и хотя бы один из них — запись, это UB. Пример — выражение `i++ + i++`.
3. **Неопределённо-упорядоченные** (indeterminately sequenced): одно вычисление выполняется целиком до или после другого, но какое именно первым — не определено. Компилятор может выбрать любой порядок.

Введённая в C11 модель sequenced-before в неизменном виде унаследована последующими редакциями стандарта C17 и C23: соответствующие разделы воспроизведены дословно. Далее в работе модель связывается со стандартом C11, имея в виду применимость и к C17, и к C23.

В языке C++ модель упорядоченности устроена близко к C11, но детализирована сильнее и различается между редакциями (так, с C++17 [7] правый операнд = вычисляется до левого, тогда как в Си операнды = не упорядочены). Настоящая работа ограничивается языком Си.

1.1.3 Неопределённо-упорядоченные вычисления

Многие операторы языка Си (`+`, `*`, `^` и др.) не вводят упорядоченности между вычислениями своих операндов. Вызов функции — исключение. Стандарт C11 (§6.5.2.2, параграф 10) устанавливает: между вычислением аргументов (включая само функциональное выражение) и входом в тело функции существует точка следования, а тело вызванной функции неопределённо-упорядочено (indeterminately sequenced) относительно всех остальных вычислений вызываемого выражения. Из этого следует, в частности, что исполнение тел функций не пересекается.

Из вышесказанного следует, что, например, в выражении `f() + g()` тела функций `f` и `g` будут выполнены целиком одно до другого. Такая конструкция не является UB, однако порядок вызова функций остаётся неопределённым. Если обе функции модифицируют общее состояние, результат может зависеть от выбранного порядка. Рассмотрим простой пример:

```
int x = 1;
int f() { x += 1; return x; }
int g() { x *= 2; return x; }
int main() { return f() + g(); }
```

Листинг 2. Зависимость результата от порядка вычисления

Если `f` вычисляется первой, `x` = 2, затем 4, результат — `2 + 4 = 6`. Если первой вычисляется `g`, `x` = 2, затем 3, результат — `3 + 2 = 5`.

Подобные конструкции встречаются и в реальном коде. Пример из интерпретатора Chibi Scheme (файл `eval.c`, функция `sexp_load_standard_env`, листинг 3):

```
sexp_env_define(ctx, ..., sym=sexp_intern(ctx, ...),
                           tmp=sexp_c_string(ctx, ...));
```

Листинг 3. Неопределённо-упорядоченные вычисления в Chibi Scheme

Функции `sexp_intern` и `sexp_c_string` аллоцируют новые объекты в куче контекста `ctx`. Оба вызова являются аргументами `sexp_env_define` и неопределённо-упорядочены между собой. Порядок, в котором они обратятся к аллокатору, зависит от компилятора. В результате `sym` и `tmp` могут получить разные указатели от аллокатора.

Формальной ошибки здесь нет: оба порядка допустимы. Однако такая непредсказуемость вредна на практике — например, если полученные от аллокатора указатели служат ключами в ассоциативном контейнере, смена порядка вызовов меняет порядок обхода и может влиять на воспроизводимость результатов.

Другая потенциальная проблема заключается в том, что функциональные макросы не порождают таких же барьеров, как обычные функции. Если в примере из листинга 2 заменить функции `f()` и `g()` на макросы, код станет иметь неопределённое поведение.

Таким образом, оба вида — как неупорядоченные модификации (UB), так и неопределённо-упорядоченные вычисления с общим состоянием — представляют собой зависимость от порядка вычислений и являются нежелательными в коде.

Дальнейшее изложение построено следующим образом. В разделе 2 рассматриваются существующие инструменты поиска подобных ошибок и их ограничения. В разделе 3 строится математическая модель отношения sequenced-before, допускающая эффективную динамическую проверку, и описывается архитектура санитайзера. Раздел 4 посвящён реализации инструмента на основе GCC, обработке крайних случаев и экспериментальной проверке на дистрибутиве Alpine Linux. В разделе 5 подводятся итоги и намечаются направления дальнейшей работы.

2. Обзор существующих решений

2.1 Статические инструменты

Существуют инструменты, позволяющие обнаруживать ошибки неупорядоченного доступа статически, на этапе компиляции. К ним относятся:

1. `-Wsequence-point` — встроенное предупреждение компилятора GCC [8], включённое по умолчанию при использовании флага `-Wall`. Выполняет статический анализ дерева выражений.
2. **Svace** — статический анализатор исходного кода, разрабатываемый в ИСП РАН [9]. Среди множества выявляемых им дефектов присутствует и неупорядоченный доступ к переменным, обнаруживаемый по дереву выражений.

Эти инструменты успешно обнаруживают простые случаи неупорядоченных модификаций:

```
i = i++ + 1;
a[i] = i++;
```

Однако конфликты, связанные с порядком вычислений, могут возникать и в более сложных ситуациях. Например, через косвенные обращения к памяти:

```
void process(int *p, int *q) {
    *p = (*q)++;
}
```

Листинг 4. Конфликт, зависящий от значений указателей

В листинге 4 представлена функция, которая не содержит UB, если гарантируется, что `p` и `q` никогда не совпадают. Определить, выполняется ли это условие в произвольной программе — задача анализа псевдонимов (alias analysis), которая в общем случае неразрешима. Это фундаментальное ограничение подхода, которое распространяется одинаково на все статические инструменты.

Другой класс конфликтов связан с побочными эффектами вызовов функций:

```
int update(struct state *s) {
    return compute(s) + s->counter++;
}
```

Листинг 5. Конфликт через побочный эффект вызова функции

Если `compute` модифицирует `s->counter`, возникнет конфликт, обнаружение которого требует межпроцедурного анализа. Для этой задачи существуют подходы статического анализа, однако их применимость часто упирается в вычислительную сложность, которую приходится сдерживать ценой потери точности. Дополнительно ситуация осложняется необходимостью производить такой анализ в сочетании с анализом псевдонимов.

2.2 Динамические инструменты

Применение статического анализа часто оказывается сопряжено с поиском компромисса между потреблением ресурсов, ложными срабатываниями и пропусками истинных. Динамический анализ к поиску зависимости от порядка вычислений, насколько известно авторам, ранее не применялся.

Среди динамических инструментов стоит отдельно упомянуть AddressSanitizer (ASan) [3], встроенный в компиляторы GCC и Clang. ASan не применим к поставленной задаче непосредственно, однако он имеет развитую инфраструктуру для инструментации взаимодействия с памятью и работы с *теневой памятью* (shadow memory). Разрабатываемый инструмент (раздел 4) опирается на эту инфраструктуру.

Основу нового инструмента составляет модель упорядоченности, следующая стандарту C11 и пригодная для эффективной динамической проверки. Её построению посвящён следующий раздел.

3. Математическая модель и архитектура решения

Рассмотрим следующий пример:

```
int x = 1;
int f(void) { return x *= 2; }
int g(void) { return f() + x++; }
int main(void) { return g(); }
```

Листинг 6. Пример программы с ошибкой

Программа из листинга 6 содержит ошибку: инкремент внутри функции `f()` может произойти в любом порядке относительно инкремента, расположенного непосредственно в `g()`. Рассмотрим отношение sequenced-before между подвыражениями этой программы и найдём то же нарушение в терминах теории графов.

На последующих рисунках приняты следующие обозначения:

1. `VC` (value computation) — вычисление значения подвыражения;
2. `SE` (side effect) — побочный эффект (см. раздел 1.1.1);
3. стрелки — отношение sequenced-before между подвыражениями;
4. пунктирные стрелки — элементы отношения, порождённые границами атомарного исполнения тел функций (см. ниже);
5. FB_x^{in}, FB_x^{out} (function boundary) — вход в тело функции `x()` и выход из него;
6. SP_i (sequence point) — точка следования стандарта C11;
7. фигурные скобки — область вычисления соответствующего вызова функции;
8. запись `&x` в `VC`-узле обозначает вычисление адреса, по которому будет произведена запись (lvalue-операнд оператора, изменяющего `x`).

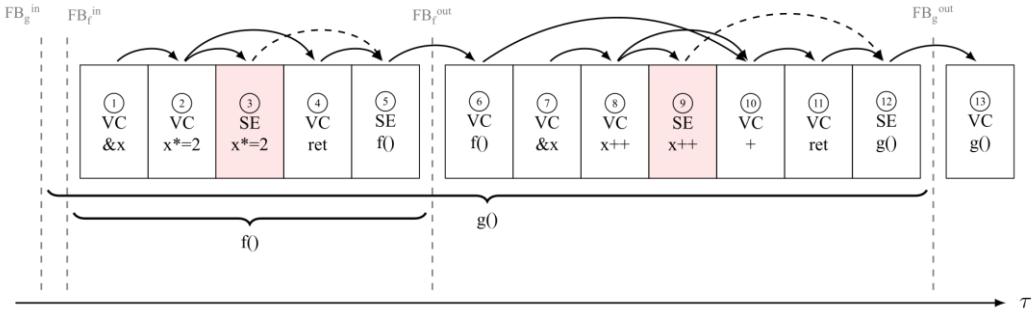

Рис. 1. Вариант исполнения программы 6

На рис. 1 изображён возможный вариант исполнения программы: компилятор вычислил тело `f` раньше, чем правый операнд сложения. Вертикальные штриховые линии FB_f^{in}, FB_f^{out}, FB_g^{in}, FB_g^{out} — границы атомарного исполнения тел функций. Согласно стандарту C11 (§6.5.2.2, параграф 10), любое вычисление в вызывающем коде, не упорядоченное явно относительно тела вызванной функции, является *indeterminately sequenced* относительно исполнения этого тела. Сноска 94 к этому же параграфу уточняет, что тело вызванной функции рассматривается как единое целое — «function executions do not interleave with each other». Формально это не точка следования, но гарантия та же: между FB^{in} и FB^{out} тело вызванной функции исполняется без прерываний со стороны вызывающего кода, и любое соседнее вычисление в вызывающем коде либо целиком предшествует ему, либо целиком следует за ним. Пунктирные стрелки на схеме выражают следствие этой атомарности: все побочные эффекты, произведённые внутри тела вызванной функции, оказываются упорядочены перед побочным эффектом охватывающего вызова. Дополнительно §6.5.16.2 и §6.5.2.4 гарантируют, что составное присваивание (*compound assignment*, `x*=2`) и постфиксный инкремент (`x++`) являются «single evaluation» относительно вызова функции, то есть не могут расщепиться через его границу.

Операции под номерами (3) и (9) — записи в `x`; между ними не существует ориентированного пути. Действуя в рамках описанных ограничений и не изменяя направления стрелок, порядок этих операций можно поменять. Такой вариант исполнения изображён на рис. 2.

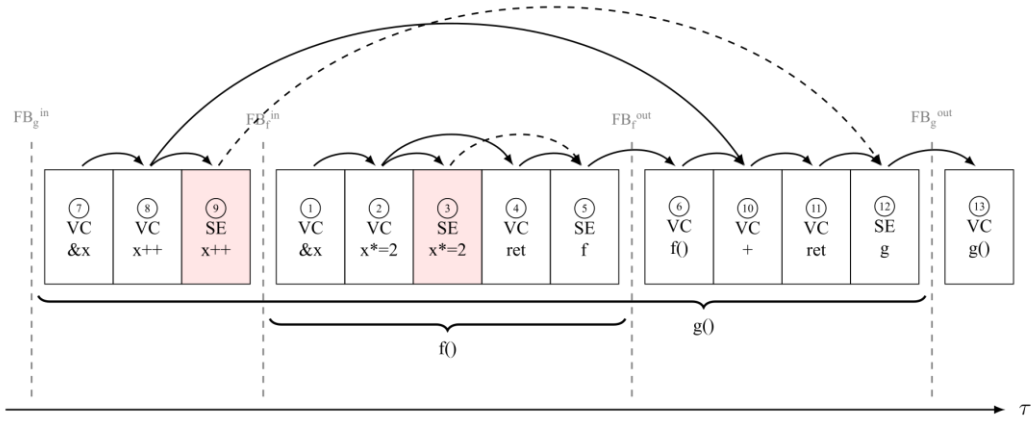

Рис. 2. Альтернативный вариант исполнения программы 6

В этом случае код возврата программы окажется равен 5, в то время как исполнение, изображённое на рис. 1, даёт результат 4. Такая перестановка оказалась возможна в связи с отсутствием цепочки, задающей строгий порядок между вычислением узлов (3) и (9).

Таким образом, интересующие нас конфликты можно выразить на языке теории графов: если два конфликтующих доступа к одной и той же переменной не имеют ориентированного пути между ними, такие доступы должны приводить к срабатыванию санитайзера.

Стоит заметить, что под такой критерий попадают и корректные программы. Например, в случае, если функция `f` производила бы инкремент вместо умножения на два, любой порядок выполнения приводил бы к одинаковому результату. На таких программах инструмент будет производить ложные срабатывания. Пример подобного ложного срабатывания на реальном коде разобран в подразделе 4.5.4.

3.1 Иерархическая структура графа sequenced-before

В исходном виде определение, данное выше, оказывается непрактичным. Общие алгоритмы поиска пути в графе требуют либо повторного обхода накопленных данных при каждом запросе, либо поддержания вспомогательных структур, ускоряющих запрос ценой расхода памяти. Если принять за n число узлов с побочными эффектами, то в обоих случаях издержки нелинейно зависят от n — суммарно по времени или по используемой памяти. Начиная с некоторого n подобный анализ оказался бы для пользователя затратнее, чем ручной поиск тех же конфликтов в коде.

3.1.1 Построение иерархической структуры

Ключевое наблюдение состоит в том, что отношение sequenced-before не является произвольным графом: на нём можно применять более эффективные алгоритмы как для хранения, так и для поиска пути между двумя узлами. Это позволяет реализовать требуемую проверку, не накапливая алгоритмическую сложность с течением времени.

Если переставить и растянуть узлы на рис. 1 согласно вложенности операций, можно увидеть, что отношение sequenced-before во многом вырождается. Большинство стрелок, обозначающих отношение $\xrightarrow[\text{S.B.}]{}$, теперь соединяет соседние узлы в схеме (см. рис. 3).

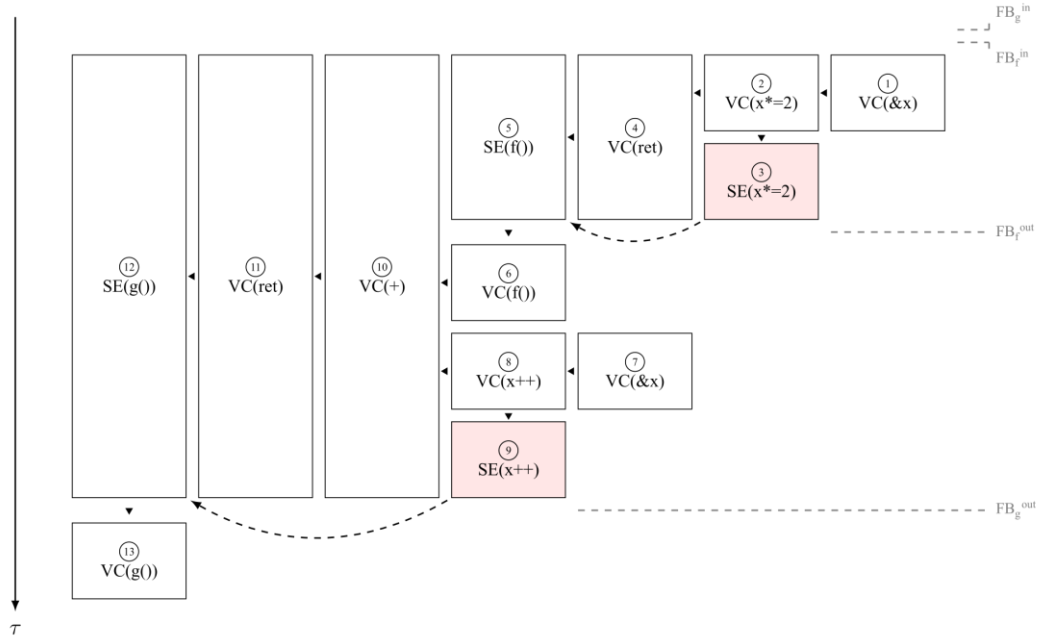

Рис. 3. Иерархическая структура графа sequenced-before для программы 6

Упростим схему далее. Рассмотрим `SE` некоторого подвыражения, который упорядочен *перед* соответствующим ему `VC` — как, например, для вызова функции. `VC`-узел в такой связке представляет собой звено, не имеющее собственных побочных эффектов, но упорядоченное относительно других узлов так же, как соответствующий `SE`-узел. Такие `SE`-`VC` пары можно объединять, не нарушая общую структуру графа. В нашем примере это означает, что пары `SE(f())` $\xrightarrow{\text{S.B.}}$ `VC(f())` и `SE(g())` $\xrightarrow{\text{S.B.}}$ `VC(g())` стягиваются в одиночные узлы `f()` и `g()`. Оставшиеся в схеме узлы `SE(x*=2)` и `SE(x++)` сохраняют свою `SE`-сторону отдельно от `VC` — по C11 (§6.5.16.2 и §6.5.2.4) их побочный эффект упорядочен *после* вычисления значения, и они не участвуют в стягивании. Будем называть такие `SE`-узлы *нестянутыми*.

Помимо общего правила, для компактности схемы удобно сделать ещё две локальные правки:

1. `VC(ret)` — не несёт собственного эффекта и служит лишь промежуточным звеном возврата из функции. Безопасен к стягиванию с нижестоящим узлом.
2. Узел `VC(+)` не требует явного префикса `VC`: оператор `+` не имеет побочного эффекта, и соответствующего узла `SE(+)` не существует, и различение `VC`/`SE` здесь излишне.

Помимо этих преобразований, на рисунке 4 и далее узлы перенумерованы в порядке pre-order DFS от корня $g()$. Эта нумерация будет полезна в дальнейшем.

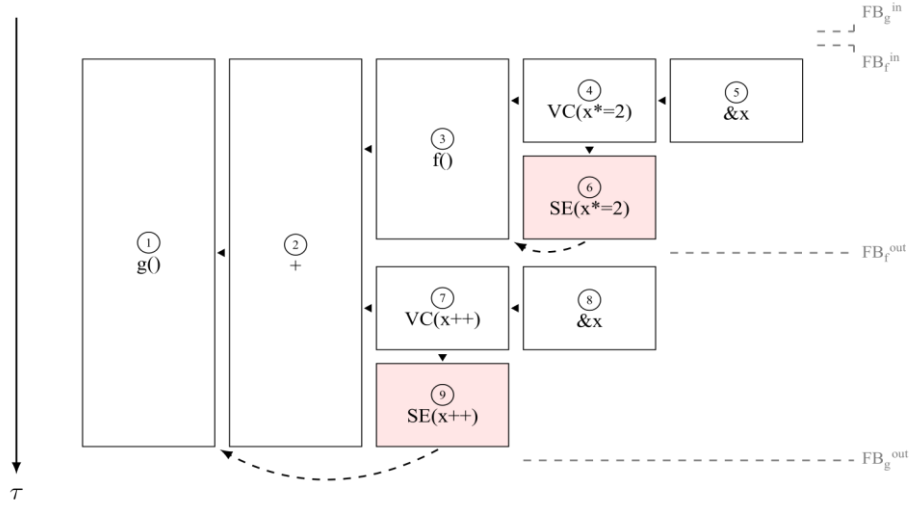

Рис. 4. Упрощённая иерархическая структура графа sequenced-before для программы 6

Схема на рис. 4 представляет собой структуру, практически повторяющую дерево подвыражений целевой программы, но с дополнительной информацией, достаточной для автоматизированного решения поставленной задачи. Отметим, что среди `SE`-узлов в такой схеме остались только нестянутые — это наблюдение будет использовано ниже при формулировании правил классификации рёбер.

3.1.2 Точки следования и классификация рёбер

Рассмотрим две функции, модифицирующие общую переменную:

```
int x = 1;
int f(void) { return x *= 2; }
int g(void) { return x *= 3; }
```

Листинг 7. Две функции с побочным эффектом на общий объект

Сравним схемы, порождаемые двумя выражениями — `f() + g()` и `f(), g()` (рис. 5).

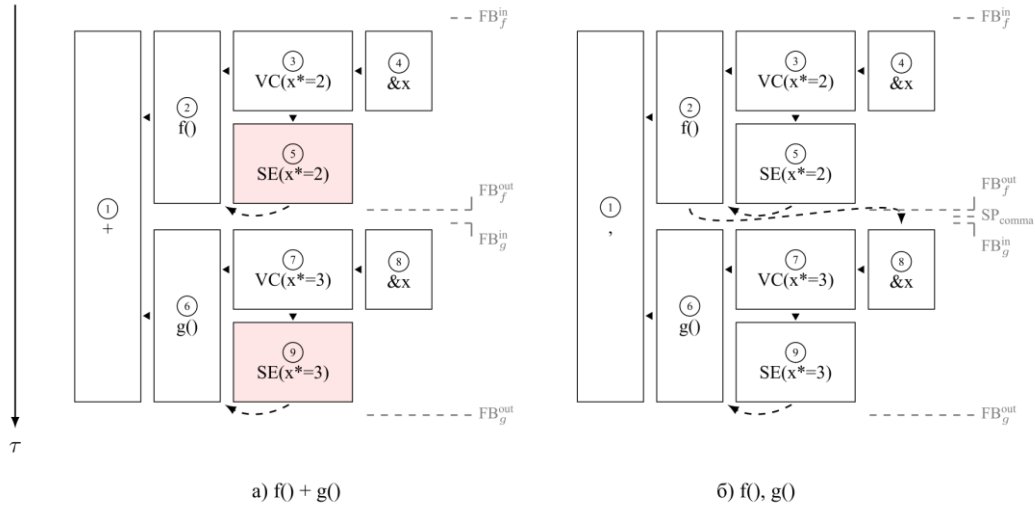

Рис. 5. Схемы для выражений над функциями из листинга 7

Левая схема — знакомый случай: оператор + не упорядочивает свои операнды, и два конфликтующих побочных эффекта `SE(x*=2)` и `SE(x*=3)` оказываются без ориентированного пути между ними.

В правой схеме между `f()` и `g()` появляется новая горизонтальная линия $\mathrm{SP}_{\mathtt{comma}}$ — точка следования, введённая оператором-запятой. Она упорядочивает каждое вычисление левого поддерева перед каждым вычислением правого. На рисунке изображено транзитивное сокращение этого набора рёбер — одна пунктирная стрелка из корня левого поддерева (2) в самый ранний узел правого поддерева (8). Теперь между `SE(x*=2)` и `SE(x*=3)` существует ориентированный путь: (5) $\xrightarrow{\text{S.B.}}$ (2) $\xrightarrow{\text{S.B.}}$ (8) $\xrightarrow{\text{S.B.}}$ (7) $\xrightarrow{\text{S.B.}}$ (9).

Выше были рассмотрены все возможные источники элементов отношения $\xrightarrow{\text{S.B.}}$ в иерархической структуре. Зафиксируем их классификацию:

1. *Child* $\xrightarrow{\text{S.B.}}$ *Parent:* любой узел, кроме нестянутых `SE`, $\xrightarrow{\text{S.B.}}$ своего родителя в иерархии.
2. *Sequence points:* точка следования в терминах C11, разделяющая поддеревья A и B. Гарантируется, что любой узел из A $\xrightarrow{\text{S.B.}}$ любой узел из B.
3. *Нестянутые side effects:* нестянутый узел `SE` некоторого подвыражения $\xrightarrow{\text{S.B.}}$ соответствующий ему `VC`.

3.2 Упрощённая модель

В качестве первого приближения построим модель, учитывающую только первые два перечисленных правила — *Child* $\xrightarrow{\text{S.B.}}$ *Parent* и *Sequence points*. Будем называть такую модель *упрощённой*. Нестянутые `SE` будут введены в модель отдельно, в разделе 3.4.

Отказавшись от нестянутых `SE`, мы избавляемся от единственного типа узлов, которые отличали иерархию из раздела 3.1 от обычного дерева подвыражений исходной программы. Теперь иерархия упрощённой модели может быть построена непосредственно из дерева подвыражений. Небольшие отступления от этой структуры будут оговариваться по мере необходимости. Каждый узел соответствует подвыражению, рёбра — отношению «подвыражение является непосредственным операндом», а точки следования размещаются между детьми соответствующих операторов. Зафиксируем модель формально.

3.2.1 Определения и обозначения

Деревом подвыражений программы назовём конечное корневое дерево $T = (V, E)$, отражающее структуру вычислений конкретного запуска: вершины V — выполненные вычисления подвыражений (одно подвыражение, исполненное несколько раз, даёт несколько вершин), рёбра E — отношение «непосредственный операнд» (родитель→ребёнок). *Поддерево* $\mathcal{T}(v)$ — это v со всеми потомками. Узлы получают *DFS-нумерацию* $\mathrm{In}\colon V \to \mathbb{N}$ — номер при pre-order обходе T от корня в порядке исполнения, выбранном компилятором; обозначим $\mathrm{Out}_v = \max\{\mathrm{In}_u \mid u \in \mathcal{T}(v)\}$. Тогда $u \in \mathcal{T}(v) \iff \mathrm{In}_v \le \mathrm{In}_u \le \mathrm{Out}_v$ — это позволяет проверять «предок-потомок» сравнением двух чисел. Множество *точек следования* $\mathrm{SP} \subseteq V \times V$ состоит из пар детей одного родителя, между которыми по C11 лежит точка следования.

Определение 1 (упрощённая модель). *Упрощённая модель* исполнения программы — кортеж $\mathcal{M} = (T, \mathrm{In}, \mathrm{SP})$. Отношение *sequenced-before* $\xrightarrow{\text{S.B.}}$ в ней — транзитивное рефлексивное замыкание объединения двух правил: *Child* $\xrightarrow{\text{S.B.}}$ *Parent* R_1 (каждый ребёнок упорядочен перед родителем) и *Sequence points* R_2 (поддерево левого участника SP-пары целиком упорядочено перед поддеревом правого):

$$R_1 = \{(c, p) \mid (p, c) \in E\},$$
$$R_2 = \{(x, y) \mid \exists\, (u, v) \in \text{SP},\ x \in \mathcal{T}(u),\ y \in \mathcal{T}(v)\}.$$

Узлы a, b *упорядочены*, если $a \xrightarrow{\text{S.B.}} b$ или $b \xrightarrow{\text{S.B.}} a$.

Через $\text{LCA}(a_1, \dots, a_k)$ обозначаем наименьший общий предок — самый глубокий узел, поддерево которого содержит все аргументы. Нам понадобятся его стандартные свойства:

1. ассоциативность $\text{LCA}(a_1, \dots, a_k) = \text{LCA}(\text{LCA}(a_1, \dots, a_{k-1}), a_k)$;
2. если $L = \text{LCA}$ не совпадает ни с одним аргументом, то какие-то два из них лежат в разных детях L;
3. LCA аргументов из $\mathcal{T}(v)$ сам лежит в $\mathcal{T}(v)$.

3.2.2 Критерий упорядоченности

Первый из двух ключевых результатов раздела — следующий критерий упорядоченности через LCA. Совместно с эффективным алгоритмом поиска LCA во время исполнения, рассматриваемым в подразделе 3.2.3, он превращает абстрактное определение sequenced-before в проверку, выполнимую за константное время.

Утверждение 1 (критерий упорядоченности в упрощённой модели). Пусть $\mathcal{M} = (T, \text{In}, \text{SP})$ — упрощённая модель. Узлы $a, b \in V$ упорядочены тогда и только тогда, когда выполнено одно из условий:

1. $\text{LCA}(a, b) = a$ или $\text{LCA}(a, b) = b$ — один из узлов является предком другого;
2. существует *цепочка точек следования* — последовательность детей $\text{LCA}(a, b)$: $c_1, c_2, \dots, c_n \in V$ ($n \geq 2$), для которой $(c_i, c_{i+1}) \in \text{SP}$ для всех $i = 1, \dots, n-1$, причём $a \in \mathcal{T}(c_1)$, $b \in \mathcal{T}(c_n)$ (или симметрично).

Содержательно критерий говорит, что два узла упорядочены ровно в двух случаях: когда один лежит в поддереве другого (правило *Child* $\xrightarrow{\text{S.B.}}$ *Parent*), либо когда их разделяет цепочка точек следования между детьми общего предка (правило *Sequence points*). Иных источников порядка в модели нет, поэтому при отсутствии такой цепочки между разными детьми $\text{LCA}(a, b)$ узлы a и b остаются неупорядоченными.

Доказательство. Достаточность. В случае (1) пусть, без ограничения общности, $\text{LCA}(a, b) = a$. Тогда b — потомок a, и применение правила R_1 вдоль пути от b к a даёт цепочку $b \xrightarrow{\text{S.B.}} \dots \xrightarrow{\text{S.B.}} a$.

В случае (2) пусть $L = \text{LCA}(a, b)$, и пусть $c_1, c_2, \dots, c_n$ — цепочка точек следования из условия. Для каждой пары $(c_i, c_{i+1}) \in \text{SP}$ правило R_2 даёт $x \xrightarrow{\text{S.B.}} y$ для всех $x \in \mathcal{T}(c_i)$, $y \in \mathcal{T}(c_{i+1})$. Применяя транзитивность, получаем

$$a \xrightarrow{\text{S.B.}} c_2 \xrightarrow{\text{S.B.}} c_3 \xrightarrow{\text{S.B.}} \dots \xrightarrow{\text{S.B.}} c_{n-1} \xrightarrow{\text{S.B.}} b,$$

откуда $a \xrightarrow{\text{S.B.}} b$.

Необходимость. Пусть $L = \text{LCA}(a, b)$, $L \neq a$, $L \neq b$, и среди детей L нет цепочки SP, связывающей поддерево, содержащее a, с поддеревом, содержащим b. Покажем, что в этом случае не выполнено ни $a \xrightarrow{\text{S.B.}} b$, ни $b \xrightarrow{\text{S.B.}} a$.

Поскольку $L \neq a$ и $L \neq b$, узлы a и b лежат в поддеревьях $\mathcal{T}(c_a)$ и $\mathcal{T}(c_b)$ двух *разных* детей c_a, c_b узла L, причём по стандартному свойству корневого дерева $\mathcal{T}(c_a) \cap \mathcal{T}(c_b) = \emptyset$.

Разделим V на три зоны: *нижнюю* $V_{\text{low}} = \bigcup_{(L,c) \in E} \mathcal{T}(c)$, сам узел L и *верхнюю* V_{up} — предки L, исключая сам L. Внутри V_{low} выделим попарно непересекающиеся поддеревья $\mathcal{T}(c_1), \mathcal{T}(c_2), \dots$ отдельных детей L. Изучим, между какими зонами идут рёбра R_1 и R_2.

Рёбра R_1 ведут из ребёнка в его родителя в T. Внутри одного поддерева $\mathcal{T}(c_k)$ они переводят узел в узел того же поддерева; на ребре (c_k, L) они выводят из V_{low} в L; начиная с L они ведут в V_{up}. В частности, ни одно ребро R_1 не переводит узел из $\mathcal{T}(c_i)$ в $\mathcal{T}(c_j)$ при $i \neq j$.

Рёбра R_2 порождаются парами $(u, v) \in \text{SP}$, причём u, v — дети одного и того же родителя. Возможны четыре случая:

1. u, v — дети самого L. Ребро R_2 ведёт из $\mathcal{T}(u)$ в $\mathcal{T}(v)$ — из одного поддерева ребёнка L в другое.
2. u, v лежат внутри какого-то $\mathcal{T}(c_k)$. Тогда $\mathcal{T}(u), \mathcal{T}(v) \subseteq \mathcal{T}(c_k)$, и ребро не покидает $\mathcal{T}(c_k)$.
3. u, v лежат в V_{up}. Тогда поддеревья $\mathcal{T}(u), \mathcal{T}(v)$ либо целиком лежат в V_{up}, либо одно из них содержит L — а с ним и оба поддерева $\mathcal{T}(c_a), \mathcal{T}(c_b)$ целиком, и сами узлы a, b. В обоих подслучаях такое ребро не соединяет a с b.
4. Один из u, v совпадает с L, а другой лежит в V_{up} (как брат L среди детей одного из его предков). Если $u = L$ и $v \in V_{\text{up}}$, ребро ведёт из $\mathcal{T}(L) = V_{\text{low}} \cup \{L\}$ в $\mathcal{T}(v) \subseteq V_{\text{up}}$ — то есть выводит из V_{low}, но не возвращает обратно. Если $u \in V_{\text{up}}$ и $v = L$, ребро ведёт из $\mathcal{T}(u) \subseteq V_{\text{up}}$ в $\mathcal{T}(L)$ — такое ребро могло бы дать переход $V_{\text{up}} \to V_{\text{low}}$, но цепочка из a доходит до V_{up} только через L и далее уже не возвращается в $\mathcal{T}(u)$ (см. разбор R_1 и предыдущие случаи R_2).

Рассмотрим произвольный путь по $R_1 \cup R_2$ из a в b. На каком-то шаге он должен покинуть $\mathcal{T}(c_a)$ и попасть в $\mathcal{T}(c_b)$. Согласно перечисленным выше свойствам, $\mathcal{T}(c_a)$ может быть покинуто либо ребром R_1 в L, либо ребром R_2 в поддерево $\mathcal{T}(c_k)$ для $c_k \neq c_a$. Из L рёбра R_1 ведут в V_{up}, а оттуда ни R_1, ни R_2 не возвращают в V_{low}. Следовательно, $\mathcal{T}(c_b)$ может быть достигнуто только последовательностью R_2-рёбер между поддеревьями детей L — иными словами, цепочкой SP-пар, ведущей от c_a к c_b. По предположению такая цепочка отсутствует, и пути из a в b не существует. □

Применим этот критерий к рис. 5. В обеих схемах рассматриваются узлы `SE(x*=2)` (5) и `SE(x*=3)` (9). Их LCA в иерархии — корневой узел (1). На левой схеме это +: оператор сложения не вводит точку следования между операндами, и узлы (5), (9) в упрощённой модели не упорядочены. На правой схеме это оператор-запятая: она разделяет своих детей через SP_{comma}, и узлы (5), (9) оказываются упорядочены.

3.2.3 Алгоритм поиска LCA во время исполнения

Чтобы алгоритм работал по мере исполнения программы без необходимости накапливать весь граф в памяти, воспользуемся уже введённой DFS-нумерацией. Дерево T и нумерация In возникают непосредственно в ходе исполнения: при входе в очередное подвыражение санитайзер создаёт соответствующий узел, присваивает ему следующий номер In_v и помещает на стек активных узлов, а при выходе из подвыражения узел снимается со стека. На любом срезе исполнения стек содержит путь от корня до текущего узла, то есть всех его предков в T. Отсюда вытекает следующий способ нахождения LCA; пример его работы приведён на рис. 6.

Утверждение 2 (поиск LCA через DFS-нумерацию). Пусть в момент исполнения узла r нам известен номер In_l некоторого ранее посещённого узла $l \in V$ (так что $\text{In}_l \leq \text{In}_r$), а стек активных узлов содержит путь $r_1, r_2, \ldots, r_n$ от корня r_1 до текущего узла $r_n = r$ — то есть $(r_i, r_{i+1}) \in E$ для всех $i = 1, \ldots, n-1$. Тогда

$$\text{LCA}(l, r) = r_k, \qquad \text{где } k = \max\{i \mid \text{In}_{r_i} \leq \text{In}_l\}.$$

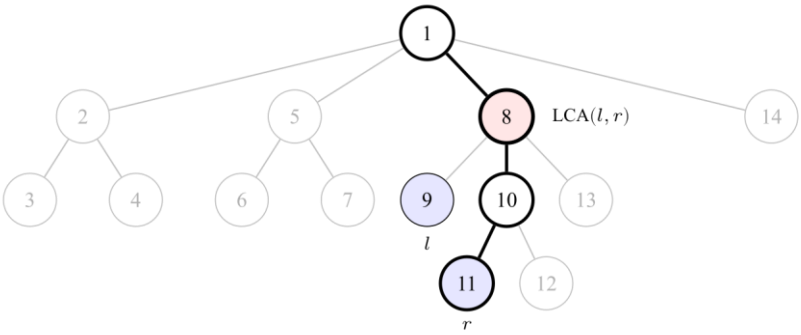

Рис. 6. Поиск LCA через DFS-нумерацию: пример при l = 9, r = 11

Доказательство. По построению, стек $r_1, \dots, r_n$ содержит всех предков r в T (включая самого r). Узел $\mathrm{LCA}(l, r)$ по определению является предком r, поэтому $\mathrm{LCA}(l, r) = r_k$ для некоторого $k \in \{1, \dots, n\}$.

По определению LCA это самый глубокий (т. е. имеющий наибольший индекс среди $r_1, \dots, r_n$) общий предок l и r. Покажем, что для любого $k \in \{1, \dots, n\}$ узел r_k является предком l тогда и только тогда, когда $\mathrm{In}_{r_k} \le \mathrm{In}_l$.

Необходимость. Если r_k — предок l (включая случай $r_k = l$), то $\mathrm{In}_{r_k} \le \mathrm{In}_l$: DFS-номер предка не превосходит номера потомка.

Достаточность. Пусть $\mathrm{In}_{r_k} \le \mathrm{In}_l$. Поскольку r_k — предок r, имеем $r \in \mathcal{T}(r_k)$, откуда $\mathrm{In}_r \le \mathrm{Out}_{r_k}$. Тогда

$$\mathrm{In}_{r_k} \le \mathrm{In}_l \le \mathrm{In}_r \le \mathrm{Out}_{r_k},$$

и по интервальному свойству DFS-нумерации $l \in \mathcal{T}(r_k)$, то есть r_k — предок l или совпадает с ним.

Итак, среди $r_1, \dots, r_n$ предками l являются ровно те r_k, для которых $\mathrm{In}_{r_k} \le \mathrm{In}_l$. Из них самый глубокий — с наибольшим индексом — и есть $\mathrm{LCA}(l, r) = r_k$, где $k = \max\{i \mid \mathrm{In}_{r_i} \le \mathrm{In}_l\}$. □

Существенно, что для нахождения $\mathrm{LCA}(l, r)$ от ранее посещённого узла l требуется единственная величина — его номер In_l. Никакой другой информации об l алгоритм не использует. Это позволяет компактно хранить состояние алгоритма: для отслеживания каждой ячейки памяти санитайзеру достаточно помнить одно число — номер последнего обратившегося к ней узла. Хранить такой номер можно в теневой памяти, привязанной к самой ячейке.

В литературе известны более эффективные алгоритмы поиска LCA, однако их детальный анализ выходит за рамки настоящей работы: уже простой линейный проход по стеку активных узлов, на котором основано утверждение 2, даёт сложность, пропорциональную глубине стека. Эта величина, в отличие от общего количества доступов к памяти, в программах на Си ограничена сверху. Тем самым проблема зависимости стоимости проверки от объёма уже выполненных обращений к памяти, сформулированная в разделе 3.1, снимается. Более того, на практике подъём по стеку почти всегда оказывается невысоким (подраздел 4.5.5), поэтому линейный проход не только достаточен по асимптотике, но и эффективен.

Заметим, однако, что производить линейный поиск по стеку следует от вершины r_n к корню r_1. В этом случае искомым r_k будет первый встретившийся узел, удовлетворяющий условию $\mathrm{In}_{r_k} \le \mathrm{In}_l$. Доступы к одной и той же ячейке памяти в реальных программах чаще находятся близко друг к другу в дереве подвыражений, поэтому их LCA лежит ближе к вершине стека, чем к корню.

Условие (2) утверждения 1 требует проверки существования цепочки точек следования между двумя детьми LCA. В общем случае ответ на этот вопрос зависит от того, какие именно дети LCA рассматриваются: один и тот же узел L может одновременно иметь и упорядоченные, и неупорядоченные пары детей. Однако всякую встречающуюся в Си конструкцию такого рода можно разбить в дереве модели на несколько узлов так, чтобы каждый оказался однородным: либо все пары его детей соединены цепочкой точек следования, либо ни одна. Конкретные примеры такой перегруппировки рассматриваются в разделе 4.2.2.

Опираясь на это наблюдение, введём следующее определение.

Определение 2 (оракул). Для модели $\mathcal{M} = (T, \mathrm{In}, \mathrm{SP})$ *оракулом* $\mathcal{O}$ называется процедура $\mathcal{O}\colon V \to \{\text{истина}, \text{ложь}\}$, такая что $\mathcal{O}(u) =$ истина, если любые два ребёнка узла u упорядочены, и $\mathcal{O}(u) =$ ложь, если никакие два ребёнка не упорядочены. Для детей одного узла упорядоченность по критерию 1 равносильна существованию цепочки точек следования между ними. Если каждый узел модели $\mathcal{M}$ попадает под одно из двух условий, будем говорить, что у модели $\mathcal{M}$ *существует оракул*. В противном случае — когда у некоторого узла одни пары детей упорядочены, а другие нет, — оракул не определён.

Описанная перегруппировка незначительно отступает от AST исходной программы: например, вершина вызова функции `CALL_EXPR` представлена двумя узлами. Внутренний узел обрамляет совокупность вычислений функционального выражения и аргументов вызова — далее будем называть её *аргументной группой*. Внешний узел обрамляет пару «аргументная группа, тело вызванной функции». В целом, однако, иерархия по-прежнему повторяет AST. Подробное обсуждение приведено в разделе 4.2.2.

Теперь объединим утверждения 1 и 2 в единый алгоритм проверки упорядоченности двух узлов: значение оракула на их LCA даёт ответ на условие (2).

Алгоритм 1 (проверка упорядоченности). Пусть известны модель $\mathcal{M}$, стек активных узлов $r_1, \ldots, r_n$ в момент исполнения текущего узла $r = r_n$ и номер In_l некоторого ранее посещённого узла l.

1. Если $\mathrm{In}_l = \mathrm{In}_r$, вернуть истину.
2. Найти $\mathrm{LCA}(l, r) = r_k$ по утверждению 2: найти первый с конца элемент r_i, удовлетворяющий $\mathrm{In}_{r_i} \le \mathrm{In}_l$.
3. Если $\mathrm{In}_{r_k} = \mathrm{In}_l$, то $r_k = l$ — узел l является предком r. Вернуть истину.
4. Иначе вернуть $\mathcal{O}(r_k)$.

Приведённый алгоритм возвращает истину, если узлы l и r упорядочены, и ложь иначе. (На шаге 3 возвращается истина в ситуации, когда l — предок r, то есть $r \xrightarrow{\text{S.B.}} l$; во всех остальных случаях, где алгоритм даёт истину, выполнено $l \xrightarrow{\text{S.B.}} r$.)

3.3 Конфликты доступов к памяти

К этому моменту построенная теория решает половину задачи: для любой пары узлов дерева подвыражений она даёт эффективный способ определить их упорядоченность. Вторая половина касается самого понятия конфликта: прежде были упомянуты конфликтующие доступы, но до сих пор их определение оставалось интуитивным. В этом разделе приводится строгое определение, а также строится алгоритм, позволяющий обнаруживать такие пары эффективно.

3.3.1 Формальное определение конфликта

Модель $\mathcal{M}$ описывает порядок исполнения подвыражений, но не сами ячейки памяти. Введём их явно. *Доступ* — тройка (v, x, t): узел $v \in V$, переменная $x \in \mathrm{Vars}$, тип $t \in \{R, W\}$ (чтение/запись); все доступы образуют множество A, а A_x — подпоследовательность

доступов к фиксированной x в фактическом порядке исполнения. Этот порядок согласован с DFS-нумерацией, но с оговоркой: доступы потомка предшествуют доступам его предка (время жизни предка охватывает жизнь потомка целиком), а доступы одного узла упорядочены порядком вычисления подвыражения и между собой конфликтовать не могут.

Стандарт C11 [5] (§6.5, параграф 2) объявляет неопределённым поведение программы, в которой «побочный эффект на скалярном объекте не упорядочен относительно другого побочного эффекта на том же объекте либо относительно вычисления значения этого же объекта». В наших обозначениях «побочные эффекты» — доступы типа W, «вычисления значения» — доступы типа R, а «unsequenced» (и «indeterminately sequenced» — постановка задачи требует покрытия обоих видов) соответствует отсутствию порядка между узлами доступов в $\mathcal{M}$. Условие на тип доступов вынесем в отдельное определение.

Определение 3 (конфликт доступов). Два доступа $a_1 = (v_1, x, t_1)$ и $a_2 = (v_2, x, t_2)$, $v_1 \neq v_2$, к общей переменной x образуют *конфликт*, если хотя бы один из них является записью: $\{t_1, t_2\} \neq \{R, R\}$.

Каждый конфликт по типам составляющих его доступов относится к одному из трёх классов: *WW* — обе стороны являются записями; *WR* — запись предшествует чтению в A_x; *RW* — чтение предшествует записи.

Условие неопределённого поведения по стандарту C11 формулируется так: доступы a_1, a_2 образуют конфликт, и при этом узлы v_1, v_2 не упорядочены в $\mathcal{M}$, то есть ни $v_1 \xrightarrow{\text{S.B.}} v_2$, ни $v_2 \xrightarrow{\text{S.B.}} v_1$. Таким образом, задача санитайзера — обнаруживать все конфликты по ходу исполнения программы и проверять узлы их доступов на упорядоченность.

В чистом виде определение 3 даёт квадратичный перебор внутри каждой последовательности A_x: каждый новый доступ-запись нужно сравнить со всеми ранее зафиксированными доступами к той же переменной. В следующем подразделе будут предложены способы сократить число таких проверок.

3.3.2 Метод критических доступов

Похожая задача уже была рассмотрена в [10] применительно к параллельной сборке: задача состояла в обнаружении конфликтующих обращений к файловой системе. Для её решения был предложен *метод критических доступов*, дающий минимальный набор проверок, гарантирующий обнаружение всех конфликтов. Число таких проверок линейно от количества доступов.

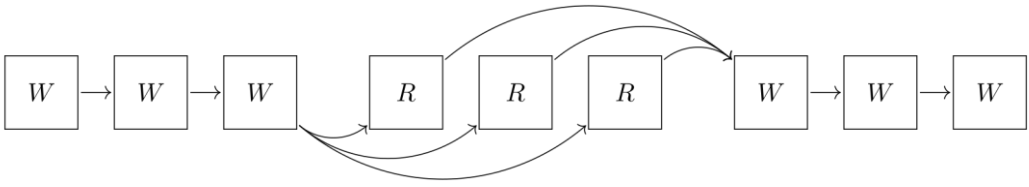

Рис. 7. Последовательность A_x и минимальный набор проверок зависимостей для метода критических доступов

В работе [10] использовались более общие обозначения — C-цепочки попарно конфликтующих доступов и N-цепочки попарно неконфликтующих, — поскольку наряду с чтением и записью в работе рассматривались дополнительные виды доступов. В настоящей работе C- и N-цепочки вырождаются в *W-цепочки* — последовательности подряд идущих записей в одну переменную — и *R-цепочки* — последовательности подряд идущих чтений между W-цепочками. Они изображены на рис. 7 группами узлов W и R. По построению W- и R-цепочки чередуются.

Алгоритм работает с потоком доступов, поступающих в реальном времени. Каждое новое чтение проверяется на упорядоченность только с последней записью. Каждая новая запись — со всеми чтениями, накопленными с момента предыдущей записи, либо, если таких чтений нет, с самой предыдущей записью.

Применить эту схему в чистом виде, однако, мешает необходимость хранить R-цепочку в динамическом массиве для каждой переменной, тогда как теневая память даёт фиксированное число бит на каждую ячейку. С другой стороны, [10] устанавливает корректность такого набора проверок без привязки к конкретной реализации. Это позволяет составить собственный алгоритм, согласованный с ограничениями теневой памяти.

Каждая проверка метода критических доступов производится между двумя доступами с общей переменной, хотя бы один из которых является записью. По типам этих доступов проверка относится ровно к одному из классов конфликтов *WW*, *WR*, *RW*. Это разбивает множество необходимых проверок C на три непересекающихся подмножества, которые далее будем обозначать C_{WW}, C_{WR} и C_{RW}: $C = C_{WW} \cup C_{WR} \cup C_{RW}$. Обозначения относятся именно к проверкам метода критических доступов, а не ко всем парам соответствующего класса: C_{RW}, в частности, — это лишь те *RW*-пары, в которых r принадлежит последней R-цепочке перед w. Ниже будут приведены два алгоритма, покрывающие $C_{WW} \cup C_{WR}$ и C_{RW} соответственно. Применённые параллельно, они охватывают всё C и тем самым обнаруживают все конфликты.

3.3.3 Алгоритм поиска *WW*- и *WR*-конфликтов

Алгоритм 2 (поиск *WW*- и *WR*-конфликтов). Для каждой переменной $x \in \mathrm{Vars}$ в теневой памяти выделяется слот S_x, хранящий In_w последней встретившейся записи w в x. До первой записи слот считается пустым. При поступлении очередного доступа $a = (v, x, t)$ алгоритм выполняет следующее:

1. Если $t = R$ и S_x непуст, выполнить проверку упорядоченности узлов w и v алгоритмом 1; состояние S_x не изменяется.
2. Если $t = W$ и S_x непуст, выполнить проверку упорядоченности узлов w и v алгоритмом 1, после чего записать $S_x \leftarrow \mathrm{In}_v$. Если S_x был пуст, проверка пропускается, а S_x инициализируется в In_v.

В обоих случаях обработка одного доступа выполняется за константное время и требует от теневой памяти лишь одного значения In на ячейку. Полный набор проверок алгоритма 2 изображён на рис. 8.

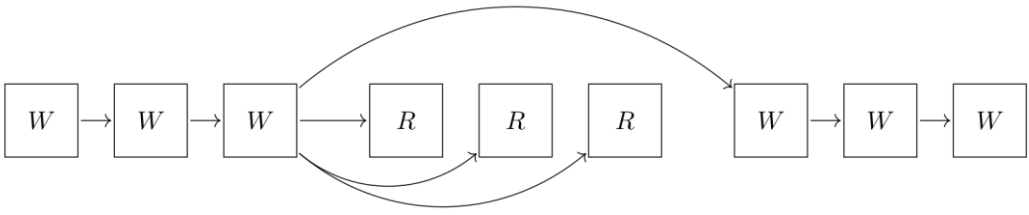

Рис. 8. Набор проверок алгоритма поиска WW- и WR-конфликтов

Покрытие алгоритмом 2 подмножества $C_{WW} \cup C_{WR}$ устанавливается следующим утверждением.

Утверждение 3 (покрытие $C_{WW} \cup C_{WR}$). Пусть для $\mathcal{M}$ существует оракул $\mathcal{O}$. Тогда множество проверок, выполняемых алгоритмом 2 на последовательности A_x, содержит $C_{WW} \cup C_{WR}$.

Обоснование прямое: слот S_x всегда хранит последнюю встретившуюся запись. Каждое новое чтение проверяется именно с ней — это проверки класса C_{WR} (от последней записи W

-цепочки к каждому чтению следующей R-цепочки); каждая новая запись проверяется с предыдущей — проверки класса C_{WW} (между соседними записями одной W-цепочки). Первая запись новой W-цепочки, приходящая после чтений, новых проверок в $C_{WW} \cup C_{WR}$ не порождает. Тем самым ни одна проверка этих классов не пропускается.

3.3.4 Алгоритм поиска *RW*-конфликтов

Оставшийся класс C_{RW} требует при каждой записи проверять её упорядоченность с каждым чтением накопленной R-цепочки. Хранить всю R-цепочку в теневой памяти невозможно. Воспользуемся следующим наблюдением: вместо того чтобы запоминать сами чтения, достаточно поддерживать единственный узел L_r — их LCA. При появлении очередного чтения r значение L_r обновляется как $L_r \leftarrow \mathrm{LCA}(L_r, r)$. Для корректности такой алгоритм должен также сбрасывать накопленный L_r, если $\mathcal{O}(L_r) =$ истина или если в A_x встречена запись.

Алгоритм 3 (поиск *RW*-конфликтов). Для каждой переменной $x \in \mathrm{Vars}$ в теневой памяти выделяется слот T_x, хранящий In_{L_r} накопленного LCA чтений текущей R-цепочки; до первого чтения после очередной записи слот считается пустым. При поступлении очередного доступа $a = (v, x, t)$ алгоритм выполняет следующее:

1. Если $t = R$ и T_x пуст, записать $T_x \leftarrow \mathrm{In}_v$.
2. Если $t = R$ и T_x непуст, вычислить $L' = \mathrm{LCA}(L_r, v)$ по стеку активных узлов согласно утверждению 2, где L_r — узел с номером T_x. Если $\mathcal{O}(L') =$ истина, записать $T_x \leftarrow \mathrm{In}_v$ (сброс), иначе записать $T_x \leftarrow \mathrm{In}_{L'}$.
3. Если $t = W$ и T_x непуст, выполнить проверку упорядоченности $L_r \xrightarrow{\text{S.B.}} v$, после чего очистить T_x. Проверка состоит в том, чтобы найти $L_{rw} = \mathrm{LCA}(L_r, v)$ по стеку активных узлов и, если $L_{rw} \neq v$ и $\mathcal{O}(L_{rw}) =$ ложь, зафиксировать конфликт.
4. Если $t = W$ и T_x пуст, ничего не делать.

Теневая память по-прежнему хранит лишь одно значение In на ячейку. Корректность алгоритма 3 устанавливается ниже через цепочку лемм. В леммах 1–4 и в утверждении 4 действуют следующие обозначения:

1. $\mathcal{M} = (T, \mathrm{In}, \mathrm{SP})$ — упрощённая модель, для которой существует оракул $\mathcal{O}$;
2. $x \in \mathrm{Vars}$ — фиксированная переменная;
3. A_x — последовательность доступов к x, к которой алгоритм 3 применяется слева направо;
4. $a_1, a_2, \ldots$ — доступы текущей R-цепочки в порядке A_x, нумерация начинается с первого доступа после последней записи (или с начала A_x, если записей не было);
5. $T_x^{(i)}$ и $L_r^{(i)}$ — состояние слота и значение накопленного LCA после обработки доступа a_i; $T_x^{(0)}$ — состояние перед первым доступом цепочки;
6. доступ $a = (v, x, t)$ в выражениях вида $a \in \mathcal{T}(X)$, In_a, $\mathrm{LCA}(a, \ldots)$ отождествляется со своим узлом v.

Промежутком накопления называется максимальный по включению отрезок текущей R-цепочки, в котором не было срабатываний шага 2 со сбросом; в пределах одного промежутка слот T_x непрерывно обновляется операциями LCA без сбросов. *Узлом сброса* называется узел L', на котором при срабатывании шага 2 значение $\mathcal{O}(L')$ оказалось истинным и привело к сбросу $T_x \leftarrow \mathrm{In}_v$.

Лемма 1 (свойство накопления). Пусть промежуток накопления состоит из доступов $a_m, a_{m+1}, \ldots, a_{m+k-1}$ $(k \geq 1)$. Тогда $L_r^{(m+k-1)} = \mathrm{LCA}(a_m, \ldots, a_{m+k-1})$. В частности, $L_r^{(m+k-1)}$ — (нестрогий) предок каждого a_j, $m \leq j \leq m + k - 1$.

Доказательство. Покажем индукцией по $j = 1, \ldots, k$, что $L_r^{(m+j-1)} = \mathrm{LCA}(a_m, \ldots, a_{m+j-1})$.

База, $j = 1$. В начале промежутка слот T_x либо был пуст (если промежуток — первый в R-цепочке), либо был сброшен предшествующим срабатыванием шага 2 (для последующих промежутков). В обоих случаях обработка a_m записывает $T_x \leftarrow \mathrm{In}_{a_m}$ (шаг 1 для пустого T_x либо ветвь сброса шага 2). Следовательно $L_r^{(m)} = a_m = \mathrm{LCA}(a_m)$.

Шаг индукции. Пусть $L_r^{(m+j-2)} = \mathrm{LCA}(a_m, \dots, a_{m+j-2})$ для некоторого $1 < j \le k$. Доступ a_{m+j-1} обрабатывается шагом 2 без сброса (по определению промежутка накопления), то есть алгоритм вычисляет $L' = \mathrm{LCA}(L_r^{(m+j-2)}, a_{m+j-1})$ и записывает $T_x \leftarrow \mathrm{In}_{L'}$. По ассоциативности LCA,

$$\mathrm{LCA}(\mathrm{LCA}(a_m, \dots, a_{m+j-2}), a_{m+j-1}) = \mathrm{LCA}(a_m, \dots, a_{m+j-1}).$$

Значит $L_r^{(m+j-1)} = \mathrm{LCA}(a_m, \dots, a_{m+j-1})$.

При $j = k$ получаем требуемое. □

Лемма 2 (промежуточные доступы лежат в $\mathcal{T}(X)$). Пусть $p, q \in A_x$, $X = \mathrm{LCA}(p, q)$. Тогда всякий доступ из A_x, появляющийся в A_x между p и q (включая сами p и q), лежит в $\mathcal{T}(X)$.

Доказательство. По определению LCA $p, q \in \mathcal{T}(X)$. Покажем от противного: если $a \notin \mathcal{T}(X)$, то a не лежит между p и q в A_x. Рассмотрим два случая.

Случай 1: a *— предок* X. Тогда a — строгий предок и p, и q, поэтому доступы p и q оба предшествуют доступам a в A_x (доступы потомка идут раньше доступов предка). Значит a не лежит между p и q.

Случай 2: a *не предок* X *и* $a \notin \mathcal{T}(X)$. Тогда a и X имеют общего предка Y, и $a \in \mathcal{T}(c_a)$, $X \in \mathcal{T}(c_X)$ для двух различных детей c_a, c_X узла Y. Так как $p, q \in \mathcal{T}(X) \subseteq \mathcal{T}(c_X)$, ни один из a, p и ни один из a, q не является предком другого. Порядок A_x для таких пар согласован с In. Поддеревья $\mathcal{T}(c_a)$ и $\mathcal{T}(c_X)$ не пересекаются, а значит, по интервальному свойству DFS-нумерации, и их In-диапазоны $[\mathrm{In}_{c_a}, \mathrm{Out}_{c_a}]$ и $[\mathrm{In}_{c_X}, \mathrm{Out}_{c_X}]$ непересекающиеся. В зависимости от их взаимного расположения In_a либо меньше обоих $\mathrm{In}_p, \mathrm{In}_q$, либо больше обоих. В первом случае a предшествует и p, и q; во втором — следует за ними. В обоих случаях a не лежит между p и q. □

Лемма 3 (L_r на цепочке предок-потомок с X). Пусть в дополнение к условиям леммы 2 p — чтение, между p и q в A_x нет записей и $\mathcal{O}(X) =$ ложь. Тогда в любой момент между обработкой p и обработкой q значение L_r лежит в том же ребёнке X, что и p, либо является нестрогим предком X.

Доказательство. По лемме 2 все доступы между p и q лежат в $\mathcal{T}(X)$. Пусть c_p — ребёнок X, содержащий p. Условие леммы перепишем как $L_r \in \mathcal{T}(c_p)$ или L_r — нестрогий предок X. Докажем его индукцией по операциям шага 2 после обработки p.

База. После обработки p по лемме 1 L_r — (нестрогий) предок p, то есть лежит на цепочке предков p (рис. 9). Эта цепочка состоит из $\mathcal{T}(c_p)$ (потомки c_p включая сам c_p) и нестрогих предков X, что и требовалось.

Шаг индукции. Пусть утверждение выполнено перед обработкой доступа a_i ($a_i \in \mathcal{T}(X)$ по лемме 2); обозначим прежнее значение через L'_r. Рассмотрим две ветви шага 2.

Ветвь без сброса устанавливает $L_r \leftarrow \mathrm{LCA}(L'_r, a_i)$. По индуктивному предположению L'_r — нестрогий предок X или $L'_r \in \mathcal{T}(c_p)$.

1. Если L'_r — нестрогий предок X, то L'_r — предок a_i, и новый $L_r = L'_r$ — по-прежнему нестрогий предок X.
2. Если $L'_r \in \mathcal{T}(c_p)$, то новый L_r — LCA узлов из $\mathcal{T}(c_p)$ и $\mathcal{T}(X)$ соответственно. Этот LCA лежит на цепочке предков X, проходящей через c_p, поэтому $L_r \in \mathcal{T}(c_p)$ или L_r — нестрогий предок X.

Ветвь со сбросом устанавливает $L_r \leftarrow a_i$, причём $\mathcal{O}(S) =$ истина для $S = \mathrm{LCA}(L'_r, a_i)$ (рис. 10). Покажем, что $a_i \in \mathcal{T}(c_p)$, разобрав два возможных положения L'_r.

1. L'_r — *нестрогий предок* X *невозможен.* Тогда L'_r — предок a_i, и $S = L'_r$. Если $L'_r = X$, то $\mathcal{O}(S) = \mathcal{O}(X) =$ ложь по условию леммы, что противоречит $\mathcal{O}(S) =$ истина. Если же L'_r — строгий предок X, то узел L'_r установился на каком-то шаге $j < i$: либо как $L_r^{(j)} = a_j \in \mathcal{T}(X)$ (по лемме 2), что несовместимо со строгим предком X; либо ветвью без сброса, условием которой было $\mathcal{O}(L_r^{(j)}) =$ ложь, что противоречит $\mathcal{O}(L'_r) = \mathcal{O}(S) =$ истина.
2. $L'_r \in \mathcal{T}(c_p)$. Тогда $S = \mathrm{LCA}(L'_r, a_i)$, где $L'_r, a_i \in \mathcal{T}(X)$; будучи LCA узлов из $\mathcal{T}(X)$, S сам лежит в $\mathcal{T}(X)$. По условию $\mathcal{O}(X) =$ ложь и $\mathcal{O}(S) =$ истина, поэтому $S \neq X$, и S — строгий потомок X. Будучи предком $L'_r \in \mathcal{T}(c_p)$, S сам лежит в $\mathcal{T}(c_p)$. Отсюда $a_i \in \mathcal{T}(S) \subseteq \mathcal{T}(c_p)$.

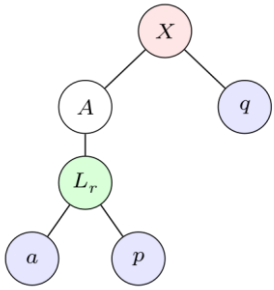

Рис. 9. Положение L_r после обработки p в текущем промежутке накопления

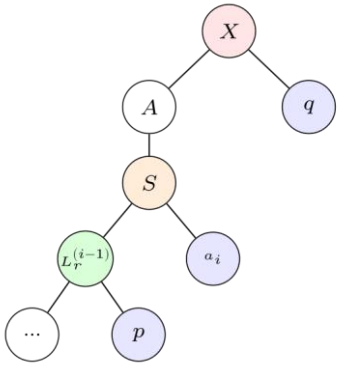

Рис. 10. Сброс на узле S после обработки p

□

Лемма 4 (оракул на финальном LCA). В условиях леммы 3 $\mathcal{O}(\mathrm{LCA}(L_r, q)) =$ ложь к моменту обработки q.

Доказательство. Между p и q в A_x записей нет, поэтому слот T_x к моменту обработки q непуст. По лемме 3 L_r лежит в том же ребёнке X, что и p, либо является нестрогим предком X. Поскольку $X = \mathrm{LCA}(p, q)$ — строгий LCA, p и q лежат в разных детях X.

Случай 1: L_r *— в ребёнке* X *с* p (см. рис. 9). Тогда L_r и q — в разных детях X, и $\mathrm{LCA}(L_r, q) = X$. По условию $\mathcal{O}(X) =$ ложь.

Случай 2: L_r *— нестрогий предок* X (рис. 11). Тогда L_r — предок q через X, и $\mathrm{LCA}(L_r, q) = L_r$. Покажем, что $\mathcal{O}(L_r) =$ ложь. Если $L_r = X$, это сразу следует из условия леммы. Иначе L_r — строгий предок X; текущее его значение установилось на каком-то i-м

шаге обработки текущего промежутка как $L_r = \mathrm{LCA}(L_r^{(i-1)}, a_i)$ через ветвь шага 2 без сброса (если бы это была ветвь сброса, L_r был бы установлен в a_i — доступ из A_x между p и q, по лемме 2 лежащий в $\mathcal{T}(X)$, что противоречит предположению о строгом предке X). Условием выбора этой ветви является $\mathcal{O}(L_r) =$ ложь.

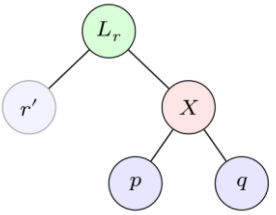

Рис. 11. L_r — предок X

□

Лемма 5 (свидетель неупорядоченности). Пусть $L_r = \mathrm{LCA}(r_{i_1}, \dots, r_{i_k})$ для некоторых узлов $r_{i_1}, \dots, r_{i_k}$, и пусть узел w не является нестрогим предком L_r и не является строгим потомком ни одного из r_{i_j}. Положим $L_{rw} = \mathrm{LCA}(L_r, w)$. Тогда найдётся r_{i_j} такой, что $\mathrm{LCA}(r_{i_j}, w) = L_{rw}$, причём r_{i_j} и w лежат в разных детях L_{rw}.

Доказательство. Поскольку w не нестрогий предок L_r, L_r либо строгий предок w, либо L_r и w не предки друг друга. Разберём оба случая.

Случай 1: L_r — строгий предок w. Тогда $L_{rw} = L_r$.

Заметим, что $L_r \neq r_{i_j}$ для всех j: в противном случае r_{i_j} оказался бы строгим предком w, что противоречит условию леммы. Поскольку L_r не совпадает ни с одним из своих аргументов, среди r_{i_j} найдутся два, лежащих в разных детях L_r. Узел w — строгий потомок L_r, поэтому лежит в одном из его детей; среди двух найденных r_{i_j} хотя бы один окажется в ребёнке L_r, отличном от ребёнка с w. Для такого r_{i_j} выполнено $\mathrm{LCA}(r_{i_j}, w) = L_r = L_{rw}$, причём r_{i_j} и w — в разных детях L_{rw}.

Случай 2: ни L_r не предок w, ни w не предок L_r. Тогда L_{rw} — строгий общий предок обоих; L_r лежит в одном его ребёнке, w — в другом. L_r — предок каждого r_{i_j}, поэтому все r_{i_j} лежат в том же ребёнке L_{rw}, что и L_r, то есть в ребёнке, отличном от ребёнка с w. Для любого r_{i_j}: $\mathrm{LCA}(r_{i_j}, w) = L_{rw}$, и r_{i_j}, w — в разных детях L_{rw}. □

Утверждение 4 (обнаружение *RW*-конфликтов). Пусть для $\mathcal{M}$ существует оракул $\mathcal{O}$. Тогда алгоритм 3 при обработке записи w фиксирует конфликт тогда и только тогда, когда среди пар $(r, w) \in C_{RW}$ найдётся такая, в которой $r \not\xrightarrow{\text{S.B.}} w$.

Доказательство. Обозначим $L_{rw} = \mathrm{LCA}(L_r, w)$.

Достаточность. Пусть $(r, w) \in C_{RW}$ и $r \not\xrightarrow{\text{S.B.}} w$. Положим $X = \mathrm{LCA}(r, w)$. Поскольку $r \not\xrightarrow{\text{S.B.}} w$, ни r, ни w не является предком другого, поэтому X — строгий LCA, и r с w лежат в разных детях X. По критерию упорядоченности (утверждение 1) $\mathcal{O}(X) =$ ложь. По определению C_{RW} доступ w — первая запись после R-цепочки, содержащей r, поэтому между r и w в A_x нет записей.

По лемме 3 (для $p := r$, $q := w$) L_r лежит в том же ребёнке X, что и r, либо является нестрогим предком X. В первом случае L_r и w — в разных детях X, $L_{rw} = X \neq w$; во втором L_r — нестрогий предок X, а значит и нестрогий предок w (через X), причём $L_{rw} = L_r$, и $L_r \neq w$, поскольку w — строгий потомок X. По лемме 4 $\mathcal{O}(L_{rw}) =$ ложь. Шаг 3 алгоритма 3 фиксирует конфликт.

Необходимость. Пусть алгоритм при обработке w зафиксировал конфликт. Тогда слот T_x непуст, $\mathcal{O}(L_{rw}) =$ ложь и $L_{rw} \neq w$. По лемме 1 $L_r = \mathrm{LCA}(r_{i_1}, \ldots, r_{i_k})$ для доступов текущего промежутка накопления.

Проверим условия леммы 5. Во-первых, w не является нестрогим предком L_r: иначе $L_{rw} = \mathrm{LCA}(L_r, w) = w$, что противоречит условию срабатывания шага 3. Во-вторых, w не является строгим потомком ни одного r_{i_j}: иначе r_{i_j} был бы строгим предком w, и доступ w предшествовал бы доступу r_{i_j} в A_x (доступы потомка идут раньше доступов предка), тогда как по предположению r_{i_j} обработан до w.

По лемме 5 найдётся r_{i_j} с $\mathrm{LCA}(r_{i_j}, w) = L_{rw}$, причём r_{i_j} и w — в разных детях L_{rw}. Из $\mathcal{O}(L_{rw}) =$ ложь по критерию упорядоченности следует $r_{i_j} \xrightarrow{\not\text{S.B.}} w$. Накопленные доступы r_{i_j} — это чтения текущей R-цепочки, идущей перед w, поэтому пара $(r_{i_j}, w) \in C_{RW}$. □

Объединяя результаты предыдущих подразделов, получаем итоговое утверждение о корректности предлагаемой схемы обнаружения конфликтов в упрощённой модели.

Утверждение 5 (обнаружение всех конфликтов). Пусть для $\mathcal{M}$ существует оракул $\mathcal{O}$. Тогда совместное применение алгоритмов 2 и 3 к последовательности A_x обнаруживает конфликт тогда и только тогда, когда в A_x существует пара неупорядоченных доступов, хотя бы один из которых — запись.

Полнота складывается из двух уже установленных фактов. Метод критических доступов [10] гарантирует, что $C_{WW} \cup C_{WR} \cup C_{RW}$ — минимальный набор проверок, эквивалентный полной таблице конфликтов: неупорядоченная конфликтующая пара в A_x существует тогда и только тогда, когда она есть среди этих трёх классов. А утверждения 3 и 4 показывают, что алгоритмы 2 и 3 вместе фиксируют конфликт ровно на этих классах. Отсюда и следует требуемая эквивалентность.

3.4 Полная модель

При построении упрощённой модели в разделе 3.2 нестянутые `SE` были сознательно исключены из рассмотрения. Настоящий раздел посвящён введению в модель недостающего правила.

3.4.1 Ограничения упрощённой модели

Рассмотрим простейший фрагмент с неопределённым поведением:

```
int a = 0;
a = a++;
```

Листинг 8. Пример программы с неопределённым поведением

В программе из листинга 8 побочный эффект простого присваивания и побочный эффект постинкремента изменяют один и тот же объект и не упорядочены между собой — по C11 (§6.5, параграф 2) поведение не определено. Тем не менее упрощённая модель этот конфликт не обнаружит. На рис. 12 изображено дерево подвыражений выражения `a = a++` с явно выписанными `VC`- и `SE`-сторонами каждого узла.

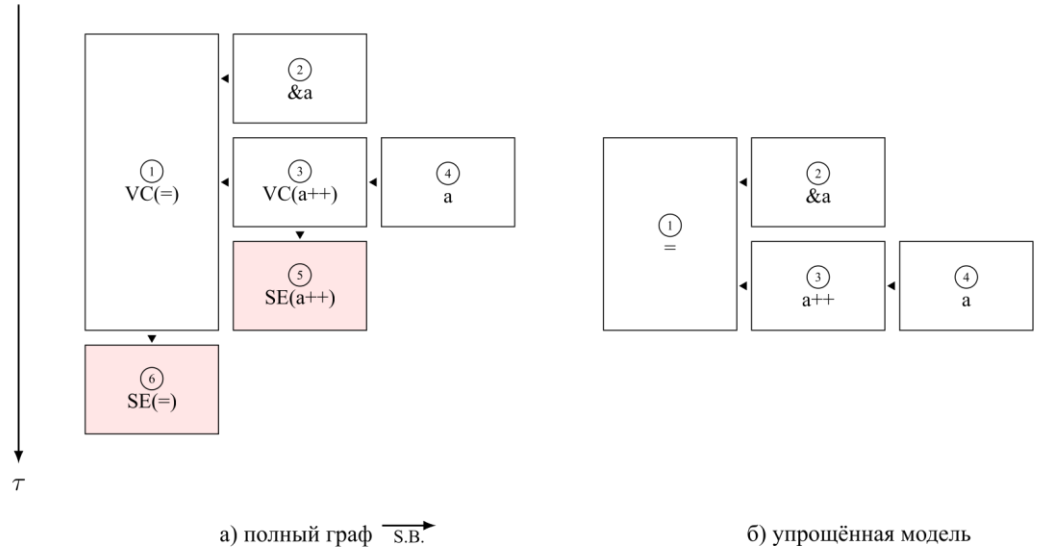

Рис. 12. Граф sequenced-before выражения `a = a++`*: полная схема с висящими* `SE`*-узлами (выделены красным) и её представление в упрощённой модели*

Согласно C11 (§6.5.16 и §6.5.2.4, параграф 2), как побочный эффект присваивания `SE(=)`, так и побочный эффект постинкремента `SE(a++)` упорядочены *после* вычисления значения соответствующих операндов. Стянуть эти `SE` с их `VC` в один узел нельзя (см. раздел 3.1.1). При переходе к упрощённой модели мы отказались от третьего правила классификации, склеив каждый нестянутый `SE` с соответствующим ему `VC` в единый узел дерева подвыражений. На рис. 12 (б) пары `VC(a++)`, `SE(a++)` и `VC(=)`, `SE(=)` превратились в одиночные узлы `a++` и = соответственно.

Такое склеивание маскирует целый класс конфликтов: узлы (5) и (6) на рис. 12 (а) конфликтуют в исходной схеме, но в упрощённой схеме (рис. 12 (б)) оказываются склеены с узлами (3) и (1) соответственно и через правило *Child* $\xrightarrow{\text{S.B.}}$ *Parent* попадают в отношение sequenced-before. Построим расширение упрощённой модели, покрывающее и этот класс конфликтов: вернём нестянутые `SE` в модель в качестве самостоятельных узлов и расширим правила классификации соответствующим образом. Будем называть такую модель *полной*.

3.4.2 Узлы SE и EB

Возвращая нестянутые `SE` в модель, мы выделяем в дереве подвыражений два специальных класса узлов.

Определение 4 (`SE`-узлы). $V_{\text{SE}} \subseteq V$ — выделенное подмножество узлов дерева подвыражений; элементы V_{SE} называются `SE`-*узлами* и соответствуют нестянутым побочным эффектам подвыражений.

`SE`-узлами полной модели становятся именно те побочные эффекты, которые не подпадают под правило стягивания со своим `VC` (раздел 3.1.1), — то есть `SE`, не упорядоченные *перед* соответствующим `VC`. Стандарт C11 выделяет такие побочные эффекты у простого и составного присваивания (§6.5.16) и постфиксного инкремента и декремента (§6.5.2.4, параграф 2). По построению `SE`-узел соответствует элементарному побочному эффекту и не содержит вложенных подвыражений: всякий `SE`-узел является листом дерева подвыражений.

В разделе 3.1 уже отмечалось, что нестянутые `SE`-узлы естественно изображать рядом со своими `VC`-узлами как соседей одного родителя. Однако в таком виде они вносят в модель неоднородность: между двумя соседями (`VC` и его `SE`) добавляется ребро $\texttt{VC} \xrightarrow{\text{S.B.}} \texttt{SE}$, которое не выражается ни правилом $\mathit{Child} \xrightarrow{\text{S.B.}} \mathit{Parent}$, ни точкой следования. Оракул при этом перестал бы укладываться в допущение «узел либо упорядочивает всех своих детей, либо никого», на которое мы опирались в разделе 3.2.3. Это усложнило бы будущую программную реализацию алгоритма.

Чтобы сохранить дерево однородным, в полной модели `SE`-узел делается *ребёнком* соответствующего `VC`, а не его соседом. Тогда пары детей одного узла снова устроены одинаково (между ними либо есть SP-цепочка, либо нет). Само же упорядочение между `VC` и его `SE` требует отдельного правила: для `SE`-узлов обычное $\mathit{Child} \xrightarrow{\text{S.B.}} \mathit{Parent}$ разворачивается, поскольку нестянутые `SE` по определению упорядочены *после*, а не до соответствующих им `VC`.

Помимо этого, на нестянутые `SE`-узлы распространяется действие точек следования. Стандарт C11 (Annex C) перечисляет следующие точки следования между подвыражениями:

1. между вычислением функционального выражения и фактических аргументов вызова и самим вызовом (§6.5.2.2);
2. между вычислением первого и второго операндов логических операторов `&&` (§6.5.13), `||` (§6.5.14) и оператора-запятой `,` (§6.5.17);
3. между вычислением первого операнда условного оператора `?:` и вычислением того из второго и третьего операндов, которое реально исполняется (§6.5.15);
4. в конце полного декларатора (§6.7.6);
5. между вычислением полного выражения и следующего за ним полного выражения. К полным выражениям относятся: инициализатор, не являющийся частью составного литерала (*compound literal*; §6.7.9); выражение в выражении-операторе (§6.8.3); управляющее выражение оператора выбора (`if`, `switch`; §6.8.4); управляющее выражение оператора `while` или `do` (§6.8.5); каждое из (опциональных) выражений оператора `for` (§6.8.5.3); (опциональное) выражение оператора `return` (§6.8.6.4).

Annex C перечисляет также точки следования внутри библиотечных функций (возврат из библиотечной функции, шаги обработки спецификаторов в форматированном вводе-выводе, вызовы comparison-функции в `qsort`/`bsearch`). Эти точки следования относятся к внутреннему устройству самой библиотечной функции и в дереве подвыражений вызывающего кода не отражаются, поэтому в дальнейшем не рассматриваются.

Отдельно отметим *атомарность тела вызова* (§6.5.2.2, параграф 10 и сноска 94, подробно разобранные во вводной части настоящего раздела при обсуждении рис. 1): тело вызванной функции исполняется как единое целое относительно охватывающих вычислений. Формально это не точка следования в смысле Annex C, однако с точки зрения отношения $\xrightarrow{\text{S.B.}}$ она имеет идентичный наблюдаемый эффект. В дальнейшем мы будем пользоваться ею наравне с обычными точками следования.

Введём новый класс узлов модели.

Определение 5 (`EB`-узлы). $V_{\texttt{EB}} \subseteq V$ — выделенное подмножество узлов дерева подвыражений. Элементы $V_{\texttt{EB}}$ называются `EB`-*узлами* (*effect boundary*). Семантика `EB`-узлов задаётся правилом R_3 из определения 6: всякий узел поддерева `EB`-узла, включая нестянутые побочные эффекты, упорядочен перед самим `EB`-узлом.

По стандарту Си точка следования всегда разделяет два дерева подвыражений; первое целиком упорядочено перед вторым. Корень левого поддерева обозначим *effect boundary*. Пометка узла как `EB` фиксирует эту границу в дереве модели явно. Такая разметка теряет

информацию о правой стороне точки следования, однако этой информации в дальнейшем и не потребуется: все новые конфликты, специфичные для полной модели, происходят целиком внутри одной левой стороны (утверждение 6). Сами `EB`-узлы — это *разметка* уже существующих узлов модели. Конкретные конструкции Си, порождающие `EB`-узлы, перечислены выше как пункты 1–3 и 5; их соответствие узлам модели приведено в подразделе 4.2.2.

Для удобства последующих алгоритмов будем считать, что каждый узел из $V_{\texttt{EB}}$ имеет ровно одного ребёнка. Если у синтаксической конструкции, претендующей на роль `EB`-узла, в дереве подвыражений несколько детей (как в случае группы «функциональное выражение и аргументы»), над ней предварительно вставляется одноребёночный искусственный узел, и пометка `EB` ставится уже на него. Алгоритмы упрощённой модели проходят через такие одноребёночные узлы прозрачно: оракул на них может принимать любое значение (пар детей нет), а правило *Child* $\xrightarrow{\text{S.B.}}$ *Parent* переносит порядок без потерь. Таким образом, добавление одноребёночных предков не нарушает ни одного результата раздела 3.2.

Каждому узлу v сопоставим $\mathrm{eb}(v) \in V_{\texttt{EB}}$ — ближайший к нему `EB`-узел-предок (самый глубокий `EB`-узел, в поддереве которого лежит v).

Соберём вместе все компоненты и правила, накопленные в этом разделе.

Определение 6 (полная модель). *Полная модель* исполнения программы — кортеж $\mathcal{M} = (T, \mathrm{In}, \mathrm{SP}, V_{\texttt{SE}}, V_{\texttt{EB}})$, состоящий из дерева подвыражений T, DFS-нумерации In, множества точек следования SP и выделенных подмножеств `SE`- и `EB`-узлов. Отношение sequenced-before $\xrightarrow{\text{S.B.}} \subseteq V \times V$ в полной модели — транзитивное рефлексивное замыкание объединения трёх отношений:

1. правило *Child* $\xrightarrow{\text{S.B.}}$ *Parent* с разворотом для `SE` (`SE`-узел упорядочен *после* своего `VC`-родителя, остальные дети — перед своим родителем, как в упрощённой модели):
$$R_1 = \{(c,p) \mid (p,c) \in E,\ c \notin V_{\texttt{SE}}\} \cup \{(p,c) \mid (p,c) \in E,\ c \in V_{\texttt{SE}}\};$$
1. правило *Sequence points*, как в упрощённой модели:
$$R_2 = \{(x,y) \mid \exists\,(u,v) \in \mathrm{SP},\ x \in \mathcal{T}(u),\ y \in \mathcal{T}(v)\};$$
1. правило *EB-граница* (всякий узел поддерева `EB`-узла упорядочен перед самим `EB`-узлом):
$$R_3 = \{(s,u) \mid u \in V_{\texttt{EB}},\ s \in \mathcal{T}(u)\}.$$

Узлы a, b *упорядочены*, если $a \xrightarrow{\text{S.B.}} b$ или $b \xrightarrow{\text{S.B.}} a$.

При $V_{\texttt{SE}} = V_{\texttt{EB}} = \emptyset$ правило R_3 исчезает, разворот в R_1 не срабатывает, и полная модель совпадает с упрощённой (определение 1). Тем самым упрощённая модель оказывается частным случаем полной. Из произвольной полной модели $\mathcal{M}$ можно получить упрощённую $\mathcal{M}'$, отождествив каждую пару (`VC`, `SE`) в один узел и забыв разметку $V_{\texttt{EB}}$. Обозначим за $\pi\colon V \to V'$ соответствующее отображение узлов (на не-`SE` тождественно, `SE`-узел переводит в его `VC`-родителя). Это соответствие используется далее как способ свести проверки в полной модели к уже доказанному критерию упрощённой.

Покажем работу `SE`- и `EB`-узлов на примере (рис. 13).

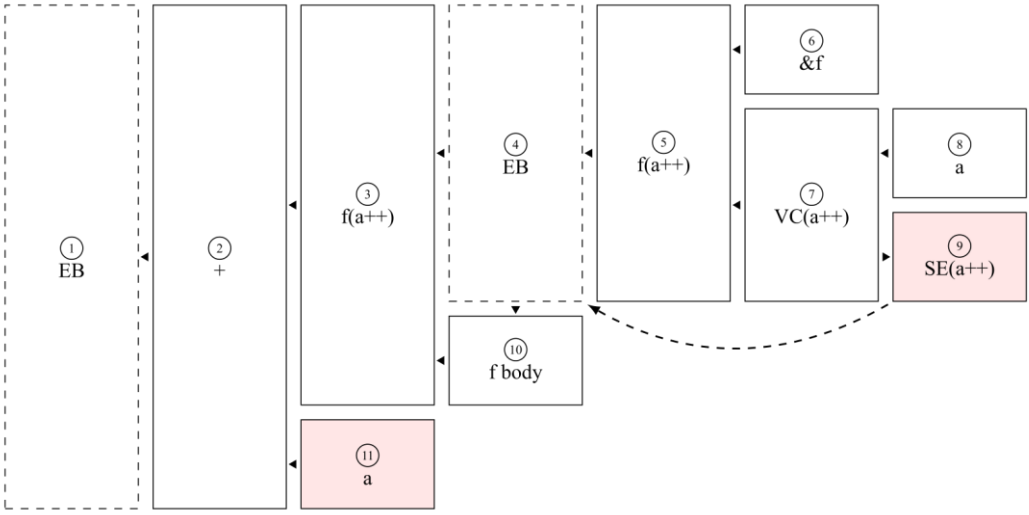

Рис. 13. Дерево полной модели для выражения `f(a++) + a`: `EB`-узел над функциональным выражением и аргументом вызова

Рассмотрим выражение `f(a++) + a`. `EB`-узел (4) помещён над аргументной группой вызова (5), объединяющей функциональное выражение (6) и аргумент (7). В этом случае `EB`-узел моделирует заявленное стандартом упорядочивание корректно: ребро (9) $\xrightarrow{\text{S.B.}}$ (4) даётся правилом `EB`-узла (побочный эффект не пересекает свою границу), а ребро (4) $\xrightarrow{\text{S.B.}}$ (10) — точкой следования из C11 (§6.5.2.2, параграф 10: «there is a sequence point after the evaluations of the function designator and the actual arguments but before the actual call»). Это даёт $\mathcal{O}((3)) =$ истина: цепочка точек следования между детьми узла (3) — между (4) и (10) — существует. Иными словами, желаемое поведение достигнуто: все побочные эффекты, относящиеся к вычислению вызываемой функции и её аргументов, оказались sequenced before самого вызова.

В этом же примере виден важный частный случай. Нестянутый `SE(a++)` (9) лежит внутри `EB`-узла (4), а второй участник конфликта — чтение `a` (11) — вне него: $\mathrm{eb}((9)) = (4)$, тогда как $\mathrm{eb}((11)) = (1)$, где (1) — `EB`-узел объемлющего полного выражения. Если склеить (9) с (7) и привести таким образом модель к упрощённой форме, то подобный конфликт уже обнаружится и без введения новых критериев — LCA пары становится (2), и критерий упрощённой модели даёт неупорядоченность. Зафиксируем это наблюдение как общее утверждение.

Утверждение 6 (покрытие трансграничных конфликтов). Пусть $u, v \in V$, $u \neq v$, $\mathrm{eb}(u) \neq \mathrm{eb}(v)$, и хотя бы один из u, v лежит в $V_{\texttt{SE}}$. Если в полной модели u и v неупорядочены, то после отождествления пар (`VC`, `SE`) критерий упрощённой модели (утверждение 1) фиксирует неупорядоченность соответствующей пары.

Иными словами, расширение упрощённой модели до полной не приносит ничего нового для пар с разными eb — их неупорядоченность уже видна упрощённой моделью после отождествления (`VC`, `SE`). Содержательно, потенциальный новый источник упорядоченности — ребро `VC` $\xrightarrow{\text{S.B.}}$ `SE` — действует только внутри одного `EB`-узла: `SE`-узел лежит под своим `VC`-родителем, и оба попадают в общий `EB`.

Итого, дополнительная логика, отличающая полную модель от упрощённой, должна распространяться только на пары (u, v), удовлетворяющие двум условиям одновременно: $\mathrm{eb}(u) = \mathrm{eb}(v)$ и хотя бы один из узлов лежит в V_{SE}.

Вернёмся к примеру 8 из раздела 3.4, который изначально мотивировал переход к полной модели. Полное дерево выражения `a = a++` с учётом `EB`-узла полного выражения приведено на рис. 14.

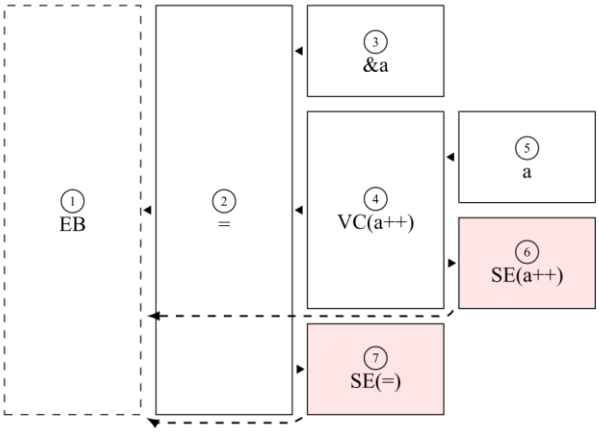

Рис. 14. Дерево полной модели для выражения `a = a++`: оба нестянутых `SE` лежат в одном `EB`-узле

Оба нестянутых побочных эффекта — `SE(a++)` (6) и `SE(=)` (7) — лежат в одном `EB`-узле (1) полного выражения: $\mathrm{eb}(6) = \mathrm{eb}(7) = 1$. Это и есть тот случай, для которого нужно расширенное правило, фиксируемое следующим критерием.

Утверждение 7 (критерий упорядоченности в полной модели). Для $u, v \in V$, $u \neq v$:

1. если $\{u, v\} \cap V_{\mathrm{SE}} \neq \emptyset$ и $\mathrm{eb}(u) = \mathrm{eb}(v)$, то u и v упорядочены в $\mathcal{M}$ тогда и только тогда, когда один из них является `VC`-родителем другого;
2. иначе u и v упорядочены в $\mathcal{M}$ тогда и только тогда, когда $\pi(u)$ и $\pi(v)$ упорядочены в $\mathcal{M}'$ по критерию упрощённой модели (утверждение 1).

Критерий покрывает ровно тот класс пар, который ранее был обозначен как нетривиальный для полной модели: $\mathrm{eb}(u) = \mathrm{eb}(v)$ и хотя бы один из узлов лежит в V_{SE}. Все остальные пары обрабатываются через отождествление (`VC`, `SE`) и критерий упрощённой модели — что согласуется с утверждением 6. Внутри же выделенного класса единственный источник упорядоченности, специфичный для полной модели и не выражающийся через упрощённую, — это прямое ребро `VC` $\xrightarrow{\text{S.B.}}$ `SE` (правило R_1 с разворотом для `SE`-ребёнка). Иных способов упорядочить пару узлов в одном `EB` полная модель не предоставляет: правило R_2 переносится в $\mathcal{M}'$ без изменений, а R_3 после отождествления (`VC`, `SE`) поглощается правилом R_1 упрощённой модели. (Утверждение «всякий узел поддерева $\mathcal{T}(u)$ упорядочен перед u» в дереве без `SE`-листьев и так следует из цепочки *Child* $\xrightarrow{\text{S.B.}}$ *Parent*.) Формальное доказательство сводится к перебору правил R_1, R_2, R_3, аналогичному доказательству утверждения 1.

3.4.3 Алгоритм проверки упорядоченности в полной модели

Симметрично алгоритму 1, переведём критерий упорядоченности полной модели в проверку, выполнимую за константное время по ходу исполнения программы. Структура алгоритма повторяет утверждение 7: ветвь 1 определяется локально по `EB`-узлам и наличию `SE`-родственников, а ветвь 2 сводится к уже введённому алгоритму 1.

Чтобы сравнение $\mathrm{eb}(u) = \mathrm{eb}(v)$ выполнялось за константное время, введём вторую нумерацию, симметричную DFS-нумерации.

Определение 7 (EB-нумерация). *EB-нумерация* — отображение $\mathrm{Eb}\colon V \to \mathbb{N}$, ставящее в соответствие каждому узлу $v \in V$ номер $\mathrm{Eb}_v := \mathrm{In}_{\mathrm{eb}(v)}$ — DFS-номер его ближайшего EB-узла-предка.

По определению, $\mathrm{eb}(u) = \mathrm{eb}(v)$ тогда и только тогда, когда $\mathrm{Eb}_u = \mathrm{Eb}_v$, поэтому соответствующее условие критерия проверяется сравнением двух целых чисел.

Актуальное значение Eb_v для текущего узла v можно поддерживать по ходу исполнения программы за $O(1)$ операций на каждое вхождение в подвыражение и выход из него: при входе в очередной узел v значение Eb_v равно In_v, если $v \in V_{\mathrm{EB}}$, и наследуется от родителя v в противном случае. В дальнейшем будем считать, что для текущего узла значение Eb_v предоставляется алгоритму извне.

Расширим теневую память: каждый слот хранит тройку $(\mathrm{In}_l, \mathrm{Eb}_l, \mathrm{isSE}_l)$, где isSE_l — однобитовый флаг, указывающий, что последний обратившийся к ячейке узел является SE-узлом. К записываемому значению In_l предварительно применяем преобразование π из утверждения 7: для SE-узла в слот заносится не его собственный In, а In его VC-родителя. Информация о принадлежности исходного узла к V_{SE} сохраняется во флаге isSE_l. Значение Eb_l заносится в слот без изменений (у SE-узла и его VC-родителя Eb совпадает). Тем самым соблюдается инвариант: In_l никогда не ссылается на SE-узел. Алгоритм 1 применим к той же теневой памяти и покрывает базовые случаи; случаи же, требующие полной модели, разрешаются с помощью флага isSE_l и счётчика Eb_l.

Алгоритм 4 (проверка упорядоченности в полной модели). Пусть известны полная модель $\mathcal{M}$, стек активных узлов $r_1, \ldots, r_n$ в момент исполнения текущего узла $r = r_n$ с известными Eb_r и isSE_r, а также тройка $(\mathrm{In}_l, \mathrm{Eb}_l, \mathrm{isSE}_l)$, извлечённая из теневой памяти для ранее посещённого узла l. Алгоритм возвращает истину, если узлы l и r упорядочены в $\mathcal{M}$, и ложь иначе.

1. Если $\mathrm{In}_l = \mathrm{In}_r$, вернуть истину: после преобразования π равенство номеров означает, что последний доступ к ячейке выполнен самим узлом r либо его SE-ребёнком, — в обоих случаях узлы упорядочены.
2. Если $\mathrm{Eb}_l = \mathrm{Eb}_r$ и хотя бы один из флагов isSE_l, isSE_r истинен (ветвь 1 критерия 7): вернуть истину, если r — SE-ребёнок узла l, то есть isSE_r и $\mathrm{In}_l = \mathrm{In}_{r_{n-1}}$ (последний — VC-родитель r по стеку); иначе вернуть ложь.
3. Иначе вернуть результат вызова алгоритма 1 на In_l и текущем узле; если isSE_r, в качестве In_r передаётся In VC-родителя r.

Алгоритмы поиска конфликтов 2 и 3 остаются в целом теми же. Дополнительно требуется лишь расширить сохраняемый в теневую память набор данных. Везде, где исходный алгоритм записывал In_v в слот, теперь записывается тройка $(\mathrm{In}_{\pi(v)}, \mathrm{Eb}_v, \mathrm{isSE}_v)$, где π — преобразование из утверждения 7, а Eb_v, isSE_v — актуальные значения, поддерживаемые санитайзером по ходу исполнения.

3.5 Итоговая архитектура

В предыдущих разделах построены ключевые компоненты санитайзера:

1. алгоритмы поиска конфликтов 2 и 3, реализующие покрытие минимального набора проверок метода критических доступов (утверждение 5);
2. алгоритм 1, реализующий критерий упорядоченности упрощённой модели, и алгоритм 4, расширяющий его до полной модели;
3. теневая память, согласованная со всеми перечисленными алгоритмами.

Совместно они образуют законченную схему обнаружения конфликтов, которая и подлежит реализации. Её потоки данных представлены на рис. 15.

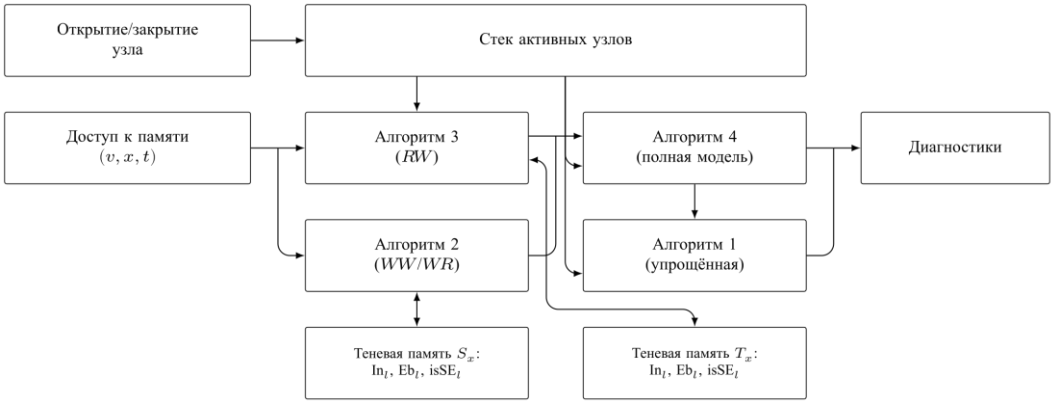

Рис. 15. Архитектура санитайзера: потоки данных между алгоритмами поиска конфликтов, проверки упорядоченности и теневой памятью

Поток входных данных — доступы к памяти, открытия и закрытия узлов — обеспечивается инструментацией санитайзера: каждое такое событие вызывает соответствующий обработчик, передающий управление коду санитайзера. Подробнее это будет описано в следующем разделе.

4. Реализация

В предыдущем разделе была построена законченная схема обнаружения конфликтов: стек активных узлов, теневая память, алгоритмы поиска *WW*/*WR*- и *RW*-конфликтов, алгоритмы проверки упорядоченности в упрощённой и полной модели. Здесь описывается её реализация в виде санитайзера для языка Си, встраиваемого в компилятор GCC.

4.1 Общая структура

Санитайзер представляет собой расширение GCC, затрагивающее четыре области его исходного кода. В каждой из этих областей санитайзер размещает либо вызовы функций, соответствующих событиям из раздела 3 (доступам к памяти, открытиям и закрытиям узлов дерева подвыражений), либо реализации самих этих функций:

1. фронтенд языка Си — сопоставление узлов абстрактного синтаксического дерева с узлами модели из раздела 3, разметка отдельных конструкций специальными признаками;
2. этап гимплификации, общий для всех языков — размещение вызовов обработчиков открытия и закрытия узлов;
3. этап преобразования промежуточного представления (*middle-end*) — новый проход `pass_seqsan` над GIMPLE, добавляющий вызовы обработчиков доступов к памяти;
4. библиотека времени исполнения `libseqsan` — часть пакета `libsanitizer`, содержащая стек активных узлов, теневую память, обработчики событий и сами алгоритмы поиска конфликтов.

Первые три области образуют *инструментирующее расширение компилятора*, четвёртая — библиотеку времени исполнения, также поставляемую в составе GCC.

Активация санитайзера производится опцией компилятора `-fsanitize=sequence`. Опция обрабатывается стандартным механизмом санитайзеров GCC и несовместима с другими санитайзерами, использующими теневую память (`address`, `hwaddress`, `thread`):

одновременное размещение в ней данных нескольких санитайзеров не представляется возможным. Конкретный способ её использования библиотекой `libseqsan` описан в подразделе 4.3.2.

4.2 Инструментирующее расширение компилятора

4.2.1 Открытие и закрытие узлов

Каждому узлу дерева подвыражений в исполняемом коде должна соответствовать пара вызовов — открывающий и закрывающий — обрамляющая исполнение его поддерева. Чтобы корректно расставить эти вызовы, санитайзеру нужно решать сразу две задачи на одном этапе: видеть исходное AST, чтобы знать тип каждого узла, и одновременно иметь возможность управлять формированием промежуточного представления — вставлять в него инструментацию, перегруппировывать вычисления, выделять побочные эффекты в отдельные поддеревья и так далее. В GCC такому совмещению удовлетворяет *гимплификация*: на ней абстрактное синтаксическое дерево исходного языка преобразуется в общеязыковое промежуточное представление GIMPLE, узлы которого напоминают команды трёхадресного ассемблера — каждый оператор содержит не более одной операции, а сложные выражения разбиваются на цепочки простых через временные переменные. До гимплификации IR ещё не существует. После неё AST исходного языка уже недоступен, и все последующие проходы средней части GCC работают только с GIMPLE.

Гимплификация позволяет фронтенду предоставить хук, через который проходит каждый узел AST; хук возвращает гимплификатору код, управляющий дальнейшей обработкой узла: `GS_UNHANDLED` (обработать общей логикой), `GS_OK` (дерево перестроено, гимплифицировать заново), `GS_ALL_DONE` (хук обработал поддерево сам) или `GS_ERROR`. Изначально контракт нужен фронтендам для подмены штатной гимплификации своих конструкций; санитайзер пользуется им, чтобы получить полный контроль над порядком обхода.

Хук санитайзера перехватывает у гимплификатора обход дерева подвыражений и выполняет его самостоятельно. Каждый вызов хука отвечает за разметку *детей* переданного ему узла: пару `__seqsan_begin`/`__seqsan_end` вокруг самого узла ставит хук, вызванный на узле-родителе. На каждом ребёнке c_i хук выполняет три действия:

1. помещает в выходную последовательность GIMPLE открывающий вызов `__seqsan_begin`;
2. вручную вызывает гимплификатор GCC на поддереве c_i с требованием материализовать его значение во временную переменную t_i;
3. помещает в выходную последовательность GIMPLE закрывающий вызов `__seqsan_end` и подменяет в AST дочерний узел c_i на t_i.

В открывающий вызов передаются тип узла модели (`U`, `S`, `EB` или `SE`; см. подраздел 4.2.2) и опционально текстовое представление узла в исходной Си-нотации (например, `"f(x, y)"`). Поскольку построение такой строки увеличивает размер кода и время компиляции, оно включается по запросу (`debug=1` в `SEQSAN_OPTIONS`); иначе подставляется заглушка `"[node]"`.

После обхода всех детей в выходной последовательности накоплен код, вычисляющий каждое подвыражение в свою временную, а сам узел в AST превратился в операцию над временными t_i. Хук возвращает `GS_OK`, сообщая гимплификатору, что дерево перестроено и работу нужно продолжить уже в новом его виде. Повторный проход общей логики приведёт AST узла к GIMPLE-представлению самой операции над временными. Пример такого преобразования для вызова `f(a(), b())` приведён на рис. 16. Иллюстрация показывает контракт хука с гимплификатором. Разметка вложенных вызовов `a()` и `b()`

(включая `EB`-обёртки и перегруппировку `CALL_EXPR`, описываемые в подразделе 4.2.2) здесь не раскрывается — она накладывается поверх той же схемы.

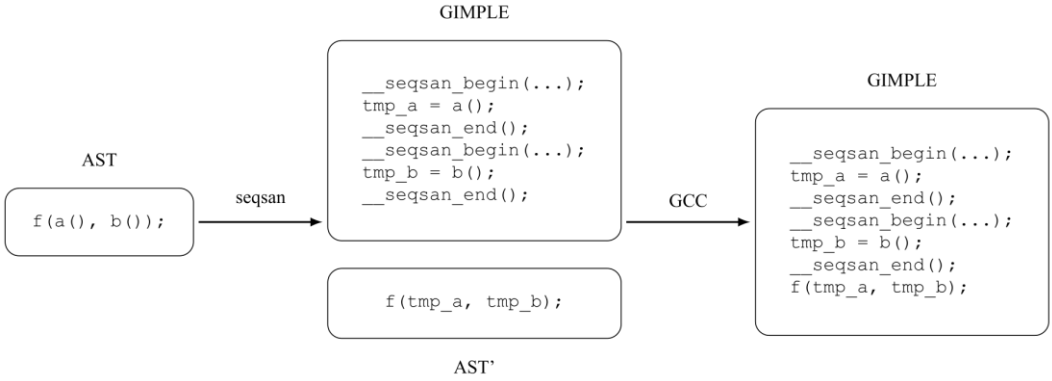

Рис. 16. Упрощённая иллюстрация взаимодействия санитайзера с гимплификатором

В случае с узлами типа `STATEMENT_LIST` возвращать `GS_OK` не требуется — эти узлы существуют как перечисления операторов, и их гимплификация сводится к последовательной гимплификации дочерних узлов, что хук и производит самостоятельно. В этом случае возвращается `GS_ALL_DONE`.

Описанный алгоритм, представляя собой значительное вмешательство в работу гимплификатора, закономерно имеет набор частных случаев, требующих отдельного внимания. Сами эти случаи и применённые к ним решения собраны в разделе 4.4.

Рекурсия по дереву получается естественным образом: вызванный санитайзером гимплификатор GCC при обработке каждого поддерева повторно вызовет тот же хук на его корне. Принудительная материализация во временную нужна для того, чтобы вычисление каждого ребёнка целиком уложилось между обрамляющими его вызовами, а не было отложено GCC до контекста, где значение фактически используется. Это даёт санитайзеру тонкий контроль над порядком исполнения и делает возможной точную расстановку вызовов `__seqsan_begin`/`__seqsan_end` вокруг каждого узла модели.

Для эффективности вложенные обрамления одного и того же типа — то есть соответствующие узлам одного класса в дереве модели, классы вводятся в подразделе 4.2.2 — объединяются. Хук поддерживает стек уже размещённых обрамлений. Если очередной открывающий вызов должен поставить узел того же типа, что уже стоит на вершине этого стека, генерация вложенного обрамления пропускается. Например, выражение $(a + b) + c$ при гимплификации без свёртки потребовало бы два вложенных одинаковых обрамления (рис. 17 (а)); после свёртки остаётся одно (рис. 17 (б)). Структуры эквивалентны: в обеих все три пары среди (a, b, c) оказываются неупорядоченными (любой LCA имеет $\mathcal{O} =$ ложь). Аналогичное рассуждение действует и для узлов с $\mathcal{O} =$ истина.

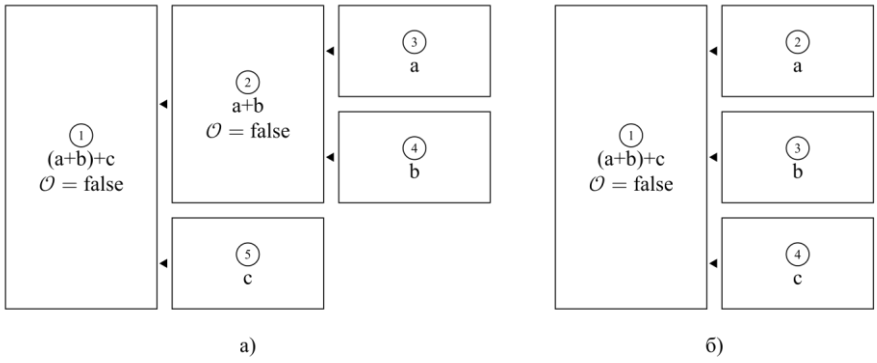

Рис. 17. Свёртка вложенных обрамлений одного типа для выражения `(a+b)+c`*: исходная двухуровневая структура (а) и результат свёртки (б)*

4.2.2 Реализация оракула

Помимо самой расстановки вызовов `__seqsan_begin`/`__seqsan_end` хук должен передавать в открывающий обработчик описание того, какому узлу модели из раздела 3 соответствует обрабатываемый узел AST. В подразделе 3.2.3 было показано, что проверка упорядоченности сводится к вычислению оракула $\mathcal{O}$ (определение 2) на узле дерева. Предположение о существовании оракула говорит, что у любого узла все пары детей ведут себя одинаково: либо все упорядочены между собой через точку следования, либо ни одна.

На чистом AST это свойство выполняется не всегда. У вызова функции вычисление функционального выражения (адреса функции) и аргументов между собой не упорядочено (C11, §6.5, параграф 3) — то есть пары вычислений внутри аргументной группы неупорядочены. После завершения аргументной группы и до входа в тело вызванной функции стандарт ставит точку следования (§6.5.2.2, параграф 10). Прямые дети одной вершины `CALL_EXPR` тем самым неоднородны: внутри аргументной группы пары вычислений неупорядочены, а пара «аргументная группа, тело» упорядочена через точку следования.

Аналогичная неоднородность возникает у составного присваивания. Вычисление левого операнда (адреса изменяемого объекта) и правого операнда между собой не упорядочено, однако чтение значения по этому адресу, выполняемое самой операцией, упорядочено после вычисления операндов и перед побочным эффектом, записывающим значение назад в переменную. Стандарт C11 определяет составное присваивание `E1 op= E2` через эквивалентность простому присваиванию `E1 = E1 op (E2)` (§6.5.16.2, параграф 3). Отсюда чтение по адресу `E1` оказывается частью вычисления правого операнда простого присваивания и по §6.5.16, параграф 3 упорядочено перед его побочным эффектом.

Подобная неоднородность в Си устраняется *перегруппировкой*: дети одной вершины AST распределяются между несколькими узлами модели так, что на каждом наборы пар детей становятся однородными. Для вызова функции это даёт два узла модели: внешний над парой «аргументная группа, тело вызванной функции», дети которого упорядочены через точку следования, и внутренний над функциональным выражением и неупорядоченными между собой аргументами. Кроме того, левое поддерево этой точки следования помечается `EB`-узлом из полной модели (определение 5) (рис. 18). Аналогичным образом расщепляется и составное присваивание. Это — упомянутые в разделе 3.2 «небольшие отступления» модели от исходного дерева подвыражений, нужные для того, чтобы оракул существовал.

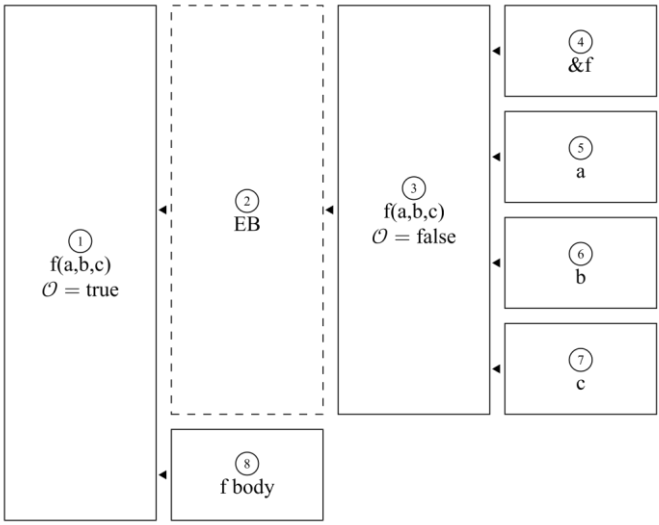

Рис. 18. Представление вызова `f(a,b,c)` в дереве модели после перегруппировки и навешивания `EB`-обёртки

После перегруппировки оракул определён на каждом узле (на листьях — тривиально) и принимает одно из двух значений. Узлы, на которых $\mathcal{O}$ = истина, будем называть `S`-*узлами* (sequenced); узлы, на которых $\mathcal{O}$ = ложь, — `U`-*узлами* (unsequenced). Помимо `U` и `S` в полной модели на отдельных узлах присутствует разметка `EB` и `SE` (определения 5 и 4). В реализации удобно объединить тип и разметку в одно поле и считать, что каждый узел стека имеет один из четырёх типов `U`, `S`, `EB` или `SE`. Этот тип хук и передаёт обработчику открытия.

Такое объединение не теряет информации. По построению полной модели `EB`-узлы одноребёночные (определение 5), а `SE`-узлы — листья (определение 4). Ни у тех, ни у других нет пар детей, и значение оракула на них в алгоритмах не используется. Если эти классы узлов занимают в перечислении типов собственные значения, ничего не нарушается — ни одно решение, принимаемое на основе этого поля, от выбора между `U` и `S` на `EB`- или `SE`-узле не зависит.

Выбор типа для каждой синтаксической конструкции Си полностью определяется её видом, что позволяет представить соответствие «вид узла → тип» в виде закрытой таблицы, прописанной во фронтенде. Опишем, какими узлами модели представляются разные виды узлов AST.

Узлы `S`.

Тип `S` присваивается узлам, у которых стандарт C11 фиксирует упорядоченность между детьми:

1. операторы `&&`, `||`, `?:` (§6.5.13, параграф 4; §6.5.14, параграф 4; §6.5.15, параграф 4);
2. внешний узел расщепления `CALL_EXPR` (§6.5.2.2, параграф 10);
3. внешний узел расщепления составного присваивания `MODIFY_EXPR` с признаком `SEQSAN_COMPOUND_ASSIGN` (§6.5.16.2, параграф 3).

Узлы `U`.

Тип `U` присваивается всем остальным внутренним узлам, у которых стандарт не вводит упорядоченности между детьми. Сюда попадают арифметические операторы (`+`, `-`, `*`, целочисленное и вещественное деление, остаток), побитовые операторы, сдвиги, операторы

сравнения, операторы индексации `a[b]` и доступа к полю `a.b`, унарные операторы, прочие конструкции без явной упорядоченности. Сюда же относятся внутренние узлы расщепления `CALL_EXPR` и составного присваивания.

Узлы `SE`.

Тип `SE` присваивается тем узлам, чьё исполнение является побочным эффектом в смысле определения 4, и соответствует трём источникам V_{SE}:

1. постфиксный инкремент и декремент — узлы `POSTINCREMENT_EXPR`/`POSTDECREMENT_EXPR`. При их гимплификации побочный эффект изменения выделяется в отдельное поддерево и оборачивается в `SE` (C11, §6.5.2.4);
2. простое присваивание (`=`) и составное присваивание (`+=`, `-=` и т.д.) — узел `MODIFY_EXPR`, побочный эффект которого по §6.5.16, параграф 3 упорядочен после вычисления значений обоих операндов. При гимплификации этот побочный эффект выделяется в отдельное поддерево и оборачивается в `SE`.

Узлы `EB`.

Для каждой точки следования стандарта Си тип `EB` присваивается одноребёночному узлу, оборачивающему её левое поддерево — то, побочные эффекты которого этой точкой следования «запечатываются». Согласно перечислению из подраздела 3.4.2, по стандарту C11 такими левыми поддеревьями являются:

1. полные выражения (§6.8, параграф 4) и их частные случаи: инициализатор, выражение в выражении-операторе, управляющие выражения `if`, `switch`, `while`, `do`, `for`, `return`;
2. левые операнды операторов `&&`, `||` (§6.5.13–6.5.14, параграф 4) и операнды оператора-запятой `,` кроме последнего (§6.5.17, параграф 2);
3. первый операнд условного оператора `?:` (§6.5.15, параграф 4);
4. аргументная группа вызова функции — внутренний узел расщепления `CALL_EXPR` (§6.5.2.2, параграф 10).

Сами операторы `&&`, `||`, `?:` при этом остаются `S`-узлами с описанными выше детьми. `EB` — только обёртка над поддеревом левого (или первого) операнда. На уровне GIMPLE полным выражениям соответствуют все вершины `STATEMENT_LIST` с побочным эффектом. Оператор-запятая в AST представлен бинарной вершиной `COMPOUND_EXPR` и в модели разворачивается в цепочку `EB`-узлов над всеми операндами кроме последнего. Последний наследует `EB` охватывающего полного выражения.

Часть конструкций порождает в AST технические узлы `COMPOUND_EXPR`, не отражающие исходный синтаксис — например, при обработке массивов переменной длины или при свёртке константных выражений. Такие узлы стандартом Си не определяются, и точки следования между их детьми не вводятся. Они помечаются признаком `SEQSAN_FORCE_UNSEQUENCED` и обрабатываются хуком как обычные `U`-узлы вместо `EB`.

4.2.3 Инструментирование доступов к памяти

Каждое чтение и каждая запись скалярного объекта в коде должны вызвать соответствующий обработчик санитайзера. Эту задачу решает проход `pass_seqsan` над промежуточным представлением GIMPLE.

Проход повторно использует код AddressSanitizer, ответственный за обход GIMPLE и выделение в нём операций доступа к памяти. Вместо встроенной функции ASan на каждом

доступе вставляется внутренний вызов `IFN_SEQSAN_CHECK`. Позднее этап `sanopt`, ответственный за разворачивание санитайзерных внутренних функций (internal functions), заменяет этот вызов на обращение к одному из обработчиков библиотеки времени исполнения:

```
__seqsan_load1,  __seqsan_load2,  __seqsan_load4,
__seqsan_load8,  __seqsan_load16, __seqsan_load_n,
__seqsan_store1, ..., __seqsan_store_n
```

выбор конкретного варианта зависит от размера доступа.

Дополнительно отключаются все алгоритмы сокращения инструментации, унаследованные от ASan. AddressSanitizer пропускает проверки, корректность которых может быть доказана статически — в первую очередь доступы к локальным переменным без взятия адреса, к статическим переменным без динамической инициализации, а также дублирующие проверки одной и той же ячейки. В настоящей работе такой статический подход не рассматривается: каждый доступ к памяти инструментируется безусловно.

Поскольку инструментируются именно обращения к памяти, скалярные переменные, которые GCC держит в SSA-форме и не материализует в памяти, из анализа выпадают: проход не встречает для них операций загрузки и сохранения. Практической ценности инструмента это не снижает. Конфликты, целиком сосредоточенные на таких переменных, поддаются статическому анализу и надёжно обнаруживаются статическим предупреждением `-Wsequence-point` (раздел 2.1). Таким образом, разрабатываемое решение представляет не замену, а расширение существующего набора инструментов.

Проход `pass_seqsan` помещён в последовательность проходов рядом с `pass_asan` и `pass_tsan` — после преобразования в SSA-форму, до удаления мёртвых записей (dead store elimination).

4.3 Библиотека времени исполнения

4.3.1 Стек активных узлов

Стек реализован поверх примитива `InternalMmapVectorNoCtor` из `sanitizer_common` — общей для всех санитайзеров инфраструктурной части `libsanitizer`. Этот примитив представляет собой динамический массив с интерфейсом, аналогичным `std::vector`, но размещаемый напрямую через системный вызов `mmap` в обход стандартного аллокатора. Обращение напрямую к `mmap` исключает зависимость библиотеки времени исполнения санитайзера от `malloc`: последний может быть перехвачен другим санитайзером либо сам по себе вносить возмущения в работу инструментированной программы. При исчерпании ёмкости буфера выделяется новый блок удвоенного размера, в который копируется содержимое старого; старый блок освобождается. Причины использования варианта `NoCtor` вместо обычного `InternalMmapVector` рассмотрены в подразделе 4.4.8. Каждый элемент стека хранит представление одного узла модели v и содержит:

1. DFS-номер узла In_v — введённая выше величина, выдаваемая монотонным счётчиком при добавлении узла на стек;
2. номер ближайшего охватывающего `EB`-узла Eb_v (определение 7), действительный на момент добавления;
3. тип узла — одно из значений `U`, `S`, `EB`, `SE` (соответствует обсуждавшемуся в подразделе 4.2.2 объединению значения оракула и разметки полной модели в одно поле); из этого поля при необходимости извлекается флаг isSE_v;
4. опционально — текстовое представление AST-узла (см. подраздел 4.2.1), используемое при отчёте о конфликте для отображения самих выражений;

5. адрес инструкции, вызвавшей добавление узла, для восстановления цепочки вызовов при сообщении о конфликте; алгоритмами поиска конфликтов этот адрес не используется.

Обработчики `__seqsan_begin` и `__seqsan_end` добавляют узел на стек и снимают его со стека. Помимо них, библиотека предоставляет два обработчика, оперирующих глубиной стека непосредственно: `__seqsan_stack_depth()` и `__seqsan_restore_depth(int)`. Они используются инструментацией вокруг `setjmp` (подраздел 4.4.1).

Монотонность счётчика, выдающего In_v, соответствует семантике этой величины из раздела 3: время добавления узла на стек полностью определяет его порядок относительно других узлов при обходе дерева в глубину. LCA двух узлов на стеке находится линейным поиском от вершины вниз до первого узла, чьё In не превосходит заданное — это и есть алгоритм из утверждения 2.

4.3.2 Теневая память

Теневая память санитайзера должна для каждой ячейки прикладной памяти, к которой возможен доступ, хранить информацию о последнем доступе к этой ячейке. Размер такой ячейки выбран равным одному байту: байт является минимальной адресуемой единицей в Си (C11, §3.6 и §6.2.6.1, параграф 2), и более мелкая гранулярность смысла не имеет. Более крупная — например, склейка двух соседних байт в общую теневую ячейку — порождала бы ложные срабатывания на соседних полях структур, где разные байты семантически независимы.

Целевой платформой реализации был выбран Linux на архитектуре x86_64. Здесь пользовательскому коду доступны 128 ТиБ виртуального адресного пространства [11]. В этих 128 ТиБ ядро и glibc размещают пользовательские отображения в трёх основных областях, разделённых обширными пустыми диапазонами. Конкретные адреса этих областей приведены в таблице 1. В свободные диапазоны между ними и впишется теневая память санитайзера.

Табл. 1. Раскладка виртуального адресного пространства пользовательских программ в Linux/x86_64

Регион	Диапазон адресов	Размер
LoAppMem	`0x000000001000 -- 0x020000000000`	2 ТиБ
свободно		83 ТиБ
MidAppMem	`0x550000000000 -- 0x5A0000000000`	5 ТиБ
свободно		32 ТиБ
HiAppMem	`0x7A0000000000 -- 0x800000000000`	6 ТиБ

Все три прикладных региона суммарно занимают 13 ТиБ, разделяющие их две свободные дыры — $83 + 32 = 115$ ТиБ. Если на каждый байт прикладной памяти теневая хранит k байт, то для всей прикладной памяти требуется $13k$ ТиБ теневой. Эти $13k$ ТиБ должны где-то разместиться в 115 ТиБ свободного места, то есть $13k \le 115$.

Из неравенства $13k \le 115$ следует $k \le 8{,}85$, то есть на каждый байт прикладной памяти доступно не более восьми байт теневой. При $k = 8$ теневая память занимает $8 \cdot 13 = 104$ ТиБ.

Оба алгоритма поиска конфликтов — *WW*/*WR* (2) и *RW* (3) — держат в теневой ячейке на байт прикладной памяти тройку $(\mathrm{In}_l, \mathrm{Eb}_l, \mathrm{isSE}_l)$, введённую в подразделе 3.4.3: два счётчика (идентификатор узла и идентификатор охватывающего `EB`-узла) и булев флаг (признак `SE`-узла). На каждую теневую ячейку при $k = 8$ приходится ровно одно 64-битное слово, в которое все три поля должны быть упакованы:

1. `is_se` (1 бит) — флаг isSE_l, признак того, что доступ выполнен внутри `SE`-узла;
2. `eb_scope` (26 бит) — величина Eb_l, идентификатор ближайшего `EB`-узла, охватывавшего этот доступ;

3. `node_id` (37 бит) — величина In_l, идентификатор узла, в котором был выполнен доступ.

Распределение бит между счётчиками асимметрично, поскольку существенно различны последствия исчерпания каждого из них. Переполнение `node_id` нарушает идентификацию узлов и поиск LCA по стеку активных узлов — корректная работа алгоритмов поиска конфликтов перестаёт быть возможной. Переполнение `eb_scope` лишь приводит к возможности коллизии `EB`-идентификаторов разных частей программы — то есть к отдельным ложным срабатываниям. Обнаружение реальных конфликтов при этом сохраняется.

Под `node_id` отводится столько бит, чтобы вероятность его переполнения на типичных запусках была пренебрежимо мала: соответствующий ему максимум $2^{37} \approx 1{,}4 \cdot 10^{11}$ узлов модели заведомо превосходит число доступов к памяти, выполняемых программой за разумное время тестирования. Оставшиеся биты отдаются `eb_scope`, и его максимум $2^{26} \approx 6{,}7 \cdot 10^{7}$ `EB`-узлов оказывается достаточен, чтобы коллизии оставались редкими.

Поддерживать оба алгоритма одновременно, однако, не представляется возможным. В той же теневой ячейке нужно было бы хранить два таких набора — то есть либо увеличить k, либо сократить разрядность счётчиков. Оба ресурса уже исчерпаны: k упирается в установленный выше потолок $k = 8$, а дальнейшее сокращение разрядности привело бы к нерационально низкому максимальному значению `node_id`. Санитайзер работает только с одним алгоритмом за раз. Выбор задаётся параметром `mode` переменной окружения `SEQSAN_OPTIONS`: по умолчанию используется *WW*/*WR*, при `SEQSAN_OPTIONS=mode=rw` обработчики переключаются на *RW*. Сама переменная `SEQSAN_OPTIONS` устроена по образцу штатных для `libsanitizer` механизмов конфигурации других санитайзеров (`ASAN_OPTIONS`, `UBSAN_OPTIONS` и так далее) и разбирается общей библиотечной инфраструктурой как список пар `имя=значение`, разделённых двоеточием.

Поведение библиотеки времени исполнения при исчерпании каждого из счётчиков соответствует их критичности, описанной выше, и настраивается параметрами `node_overflow` и `seqpt_overflow` переменной окружения `SEQSAN_OPTIONS`.

Обработчики доступов `__seqsan_load*` и `__seqsan_store*` читают теневую ячейку по адресу обращения, исполняют буквальную реализацию алгоритма 2 или 3 (в зависимости от выбранного режима) и при обнаружении конфликта выводят отчёт — идентификаторы и типы конфликтующих узлов, LCA, текущий стек активных узлов, цепочку вызовов — после чего программа по умолчанию завершается ненулевым кодом.

4.4 Обработка крайних случаев

Часть конструкций языка Си и особенностей самого GCC не укладывается в картину, описанную в подразделах 4.2 и 4.3: одни нарушают порядок исполнения, заданный деревом подвыражений, другие приводят к внутренним ошибкам компилятора, третьи остаются за пределами применимости санитайзера. Настоящий раздел собирает такие случаи и описывает их обработку.

4.4.1 Обработка длинных прыжков

Особое место среди инструментируемых конструкций занимают вызовы функций с признаком `ECF_RETURNS_TWICE`, к которым относится `setjmp`. Исполнение фрагмента кода между `setjmp` и `longjmp` может быть прервано в произвольной точке: при выходе через `longjmp` управление возвращается в место `setjmp`, но все добавления узлов на стек, выполненные между ними, остаются.

Рассмотрим характерный пример (листинг 9). В строке 9 при вызове функции `f` перед вычислением аргументов на стек добавляется узел `U`, соответствующий аргументной группе. Если внутри `worker` срабатывает `longjmp` (строка 2), управление уходит обратно к `setjmp` (строка 8), минуя соответствующий `__seqsan_end`. Этот узел `U` остаётся на стеке, и все последующие проверки в той же функции застанут его в составе цепочки активных узлов, что нарушит консистентность стека и корректность алгоритмов.

```
int worker(jmp_buf *b) {
    if (error) longjmp(*b, 1);
    return ok;
}

int main() {
    jmp_buf buf;
    if (setjmp(buf) == 0)
        f(worker(&buf), g());
}
```

Листинг 9. Прыжок `longjmp` из вычисления аргументной группы

Относительно выхода из `longjmp` стандарт даёт две гарантии, которыми санитайзер может пользоваться. Их описывает §7.13.2.1, параграф 2 — к моменту прыжка функция, содержащая соответствующий `setjmp`, должна оставаться активной, а если `setjmp` находился внутри области видимости переменно модифицируемого типа, то и эта область должна оставаться активной, иначе поведение программы не определено. Из первой гарантии следует, что к моменту возврата из функции, содержащей `setjmp`, все узлы, открытые между этим `setjmp` и соответствующим ему `longjmp`, гарантированно покинуты, и стек активных узлов снова консистентен. Вторая гарантия позволяла бы возобновлять поиск конфликтов раньше — при выходе из области видимости переменно модифицируемого типа, — но в настоящей версии санитайзера такое уточнение не реализовано. Инструментация подавляется непосредственно перед `longjmp` и остаётся выключенной до возврата из функции, содержащей `setjmp`.

Реализация работает с этим случаем через два механизма.

Первый — локальное согласование глубины стека вокруг самого `setjmp`. На стороне компилятора `pass_seqsan` непосредственно перед каждым вызовом с признаком `ECF_RETURNS_TWICE` сохраняет текущую глубину стека в локальную переменную, а сразу после вызова обращается к `__seqsan_restore_depth` от этой переменной. На обычном пути из `setjmp` глубина уже совпадает с сохранённой, и восстановление ничего не делает. При возврате через `longjmp` оно вернёт глубину к значению, сохранённому до `setjmp`.

Второй — подавление поиска конфликтов на время, пока стек ещё может содержать узлы из перезаписанных областей. На стороне библиотеки времени исполнения функции `longjmp` и `siglongjmp` перехватываются: перед вызовом оригинальной `longjmp` перехватчик устанавливает переменную `in_longjmp`, и обработчики доступов к памяти при установленном признаке пропускают обращение. Признак сбрасывается при возврате из функции, содержащей `setjmp`: на стороне компилятора `pass_seqsan` оборачивает тело такой функции в `TRY_FINALLY`, в `finally`-секцию которого вставляется обращение к обработчику, сбрасывающему `in_longjmp`. На обычном пути исполнения секция отрабатывает сразу после нормального `return`, а на пути через `longjmp` — после того, как исполнение покинет фрейм этой функции. В обоих случаях к моменту сброса стек активных узлов снова гарантированно консистентен, и поиск конфликтов возобновляется. Вложенные вызовы функций, имеющие место между `longjmp` и возвратом из функции, содержащей `setjmp`, обрабатываются отдельно и не сбрасывают `in_longjmp` раньше времени.

4.4.2 Обработка switch и goto

Помимо длинных прыжков, локальные передачи управления внутри функции — `goto`, `break`, `continue`, а также неявные переходы внутри `switch` — тоже способны нарушить порядок открытия и закрытия узлов модели. Поток управления может попасть в произвольное место среди операторов блока через метку (`LABEL_EXPR`, `CASE_LABEL_EXPR`), либо, наоборот, покинуть блок раньше, чем будут закрыты все открытые в нём узлы. В обоих случаях соответствующие `__seqsan_end` оказываются пропущены, и стек активных узлов остаётся в неконсистентном состоянии.

Решение строится на разбиении каждого `STATEMENT_LIST` на сегменты по меткам: метки становятся точками разреза, а операторы между двумя соседними метками собираются в один сегмент. Каждый сегмент оборачивается в `TRY_FINALLY` с двумя действиями. В пролог сегмента `pass_seqsan` вставляет вызов `__seqsan_stack_depth()`, сохраняющий текущую глубину стека в локальную переменную, а в `finally`-секцию — вызов `__seqsan_restore_depth` от этой переменной. При любом обычном выходе из сегмента — падении на следующую метку, `break`, `continue` — срабатывает `finally`-секция и приводит глубину стека к сохранённому значению, тем самым закрывая все узлы, открытые внутри сегмента. Сами метки выносятся между сегментами наружу, чтобы вход через `goto` или `switch` приземлялся в начало нового сегмента, а не внутрь чужой `TRY_FINALLY`-обёртки.

Для корректности схемы важно, чтобы сегменты не пересекались и чтобы ни одна метка не оказывалась внутри уже открытого сегмента — иначе вход через такую метку по `goto` или диспетчеру `switch` обошёл бы сохранение глубины в прологе сегмента, а `finally`-секция при выходе восстанавливала бы её из неинициализированной переменной. Поэтому, встречая при обходе `STATEMENT_LIST` оператор, содержащий внутри себя метку на произвольной глубине вложенности (в частности, `BIND_EXPR` и `SWITCH_EXPR`), хук закрывает текущий сегмент и помещает такой оператор в выходную последовательность отдельно, оставляя сегментацию его внутренних списков рекурсивному вызову.

Аналогично трактуется `IFN_FALLTHROUGH` — внутренняя функция, в которую GCC превращает аннотацию `__attribute__((fallthrough))` в теле `switch`. Семантически эта аннотация маркирует намеренный переход к следующей метке `case` и потому должна непосредственно ей предшествовать. Если бы хук завернул `IFN_FALLTHROUGH` в предшествующий `TRY_FINALLY`-сегмент, между аннотацией и меткой оказалась бы `finally`-секция с восстановлением глубины, и соседство было бы нарушено. Решение — `IFN_FALLTHROUGH` трактуется как единая граница сегмента вместе с самой `case`-меткой.

4.4.3 Работа с многопоточностью

Поддержка многопоточности требовала бы выполнять алгоритмы поиска конфликтов в каждом потоке независимо. Поскольку теневая память у потоков общая, в теневой ячейке пришлось бы дополнительно хранить и индекс потока, выполнившего доступ. В противном случае на переменных, к которым обращаются разные потоки, поиск LCA между двумя доступами не имел бы смысла и нарушал бы корректность алгоритмов.

Добавление в теневую ячейку какого-либо дополнительного поля в выбранной схеме теневой памяти невозможно: расширение разрядности теневой ячейки до девяти байт на байт прикладной памяти потребовало бы $9 \cdot 13 = 117$ ТиБ теневой при доступных 115 ТиБ, а уменьшение разрядности In_l и Eb_l даже на несколько бит существенно ухудшило бы устойчивость счётчиков к длительным запускам.

В связи с этим было принято консервативное решение: инструментация производится только в главном потоке программы. Любое обращение к библиотеке `libseqsan` из других потоков немедленно возвращает управление.

4.4.4 Обработка сигналов

Обработчики сигналов можно рассматривать как отдельные потоки с ещё меньшим числом гарантий: запуск асинхронный, точка прерывания произвольна, стандарт C11 не вводит между побочными эффектами обработчика и эффектами прерванного кода никакого отношения упорядоченности. По причинам, аналогичным описанным в подразделе 4.4.3, инструментация внутри обработчиков сигналов подавляется.

Технически это достигается через штатный для `libsanitizer` механизм *перехвата вызовов библиотечных функций* — общий для всех санитайзеров (ASan, TSan и других), позволяющий библиотеке времени исполнения подменять произвольную функцию libc собственной обёрткой, которая при необходимости вызывает оригинальную реализацию. Перехват устроен в два этапа. На этапе компоновки макрос `INTERCEPTOR` объявляет функцию-обёртку и создаёт для неё *слабый псевдоним* (weak alias) с именем перехватываемой функции. Благодаря этому компоновщик разрешает символ, например `sigaction`, в код санитайзера, а не в реализацию glibc. На этапе инициализации библиотеки времени исполнения макрос `INTERCEPT_FUNCTION` через `dlsym(RTLD_NEXT, …)` находит настоящий адрес оригинала в glibc и сохраняет его во внутреннем слоте. Из обёртки оригинал вызывается через макрос `REAL(…)`. В санитайзере этот механизм используется для перехвата `signal`, `sigaction`, `longjmp` и `siglongjmp` (подраздел 4.4.1). Кроме того, в усечённой форме (только слабый псевдоним, без обращения к оригиналу) он используется для подмены `vfork` в подразделе 4.4.5.

В случае сигналов перехватываются `signal` и `sigaction`: вместо переданного пользователем обработчика регистрируется промежуточная функция, которая перед его вызовом устанавливает признак `in_signal_handler`, а после — сбрасывает. При установленном признаке обработчики доступов к памяти пропускают обращение.

4.4.5 Обработка vfork

Вызов `vfork` формально имеет признак `ECF_RETURNS_TWICE`, как и `setjmp`, но по существу является оптимизированной версией `fork`: родительский процесс блокируется до завершения дочернего, а дочерний разделяет с родительским виртуальную память. Разделение памяти с дочерним процессом требует от компилятора соблюдения дополнительных правил при использовании `vfork`. Инструментация этого вызова нарушает эти правила, приводя к внутренней ошибке компилятора (ICE). `pass_seqsan` вызовы `vfork` проходом не оборачивает.

Согласованность стека и теневой памяти при `vfork` обеспечивается на стороне библиотеки времени исполнения — по образцу решения, применённого в ThreadSanitizer, перехватчик (подраздел 4.4.4) подменяет вызов `vfork` на обычный `fork`. Такая замена не нарушает корректности: `vfork` является оптимизированным вариантом `fork` с более узким набором допустимых сценариев, и любую программу, корректно использующую `vfork`, можно без последствий запустить с `fork` вместо него.

4.4.6 Перехват `__tls_get_addr`

Общая инфраструктура `sanitizer_common` перехватывает функцию динамического загрузчика `__tls_get_addr`, через которую разрешаются обращения к переменным потоково-локального хранилища (TLS), чтобы при выделении каждого нового TLS-блока вызвать общий хук `COMMON_INTERCEPTOR_INITIALIZE_RANGE`. Поведение хука определяется конкретным санитайзером. Сам перехватчик хранит данные на поток в собственной TLS-переменной, обязательно с моделью initial-exec — иначе обращение к ней само вызывало бы `__tls_get_addr` и приводило бы к бесконечной рекурсии.

Однако initial-exec-переменные не поддерживаются в библиотеках, загружаемых через `dlopen` уже после старта процесса. Такой сценарий оказался распространённым в описываемом ниже тестовом окружении (подраздел 4.5.2), где опция `-fsanitize=sequence` безусловно добавляется во все стандартные переменные окружения сборки собираемых пакетов: всякая инструментированная разделяемая библиотека, динамически загружаемая в уже работающую программу (например, как плагин для `Python` или `PostgreSQL`), загружает за собой и `libseqsan`.

`COMMON_INTERCEPTOR_INITIALIZE_RANGE` не используется в `libseqsan`: теневая память покрывает весь прикладной диапазон адресов (включая регион `HiAppMem`, в котором располагаются TLS-блоки) сразу при инициализации библиотеки, и никакой дополнительной обработки TLS со стороны санитайзера не требуется. Перехватчик подавляется на этапе компиляции через переопределение макроса `SANITIZER_INTERCEPT_TLS_GET_ADDR` в значение `0`. Это стандартная для `libsanitizer` техника, аналогично применённая в `libhwasan` для отключения `__tls_get_addr`. Реализация находится в отдельном заголовочном файле `seqsan_platform_interceptors.h`, принудительно включаемом в каждый файл при сборке. Вместе с подавлением перехватчика из библиотеки удаляется и проблемная переменная.

4.4.7 Явное переключение контекста

Функции `setcontext` и `swapcontext` стандарта POSIX выполняют переход на ранее подготовленный стек исполнения по запросу пользовательского кода. В отличие от `longjmp` это не возврат вверх по уже существующему стеку, а полная подмена точки исполнения и стека на другие. Стек активных узлов санитайзера, целиком привязанный к одному потоку исполнения, после такого переключения оказывается с ним рассогласован, что нарушает корректность алгоритмов. Поддержка этих функций невозможна по тем же соображениям, что и поддержка многопоточности (4.4.3): теневая ячейка не имеет в своём бюджете разрядов под идентификатор контекста, в котором был выполнен доступ. В текущей реализации обе функции перехватываются и при первом же вызове аварийно завершают исполнение программы.

Остальные функции того же семейства `ucontext` — `getcontext` и `makecontext` — сами по себе для санитайзера безвредны: первая лишь сохраняет текущий контекст, вторая настраивает заранее выделенный буфер под будущий переход. Никакой специальной обработки для них не требуется.

Само семейство `ucontext` помечено как устаревшее в IEEE Std 1003.1/Cor 2 [12] с рекомендацией заменять его POSIX-потоками, а в POSIX.1-2008 [13] полностью изъято из стандарта как непортируемое. Отсутствие полноценной поддержки этих устаревших функций санитайзером не ограничивает его применимости на практике.

4.4.8 Иные крайние случаи

Помимо разобранных выше, в ходе проверки санитайзера на сторонних проектах обнаружился ряд узких технических случаев, требовавших отдельного внимания. Каждый из них имел достаточно простое решение. Выделять каждую такую ситуацию в отдельный подраздел нецелесообразно — все они собраны в общую сводку ниже.

1. **Повторная гимплификация после перехода в SSA-форму.** Некоторые поздние проходы — в частности, `pass_laddress` при `-O2` — повторно запускают гимплификацию над уже обработанными подвыражениями. К этому моменту функция уже находится в SSA-форме, и повторный заход в наш хук с попыткой ещё раз выделить детей во временные нарушил бы SSA-инвариант, что приводит

к внутренней ошибке компилятора. Хук распознаёт такую ситуацию по флагу `PROP_ssa` в `cfun->curr_properties` и возвращает `GS_UNHANDLED`.

2. **Гимплификация служебных временных объявлений.** При материализации очередного ребёнка хук вызывает `gimplify_expr`, которая внутри обращается к `get_formal_tmp_var`, та строит `INIT_EXPR` для создаваемой временной переменной и снова уходит в `gimplify_stmt`. Языковой хук срабатывает на этом `INIT_EXPR`-узле повторно, попадая в бесконечную рекурсию. Все такие порождаемые компилятором временные имеют признак `DECL_ARTIFICIAL` на левом операнде `INIT_EXPR`/`MODIFY_EXPR`, и хук распознаёт их по этому признаку и пропускает без собственной разметки.
3. **Переменно модифицируемые типы.** Если очередной ребёнок имеет переменно модифицируемый тип — например, указатель на VLA — штатная `gimplify_expr` при выделении его во временную обращается к `force_constant_size`, требующей константного размера типа, и завершается с ошибкой. Хук распознаёт такие случаи по предикату `variably_modified_type_p` и пропускает ребёнка целиком: его поддерево не материализуется во временную, и пара `__seqsan_begin`/`__seqsan_end` вокруг него не ставится. Гимплификация самого ребёнка отдаётся общей логике, которая на верхнем уровне родителя достроит вычисление узла модели как операцию над его поддеревьями напрямую, без введения временных.
4. **Встроенные функции вида** `__builtin_va_arg_pack`**.** Они допустимы только как буквальный последний аргумент вызова и не могут быть заменены на временную переменную. Хук пропускает их без гимплификации.
5. **Стек активных узлов и глобальные деструкторы.** Обычный примитив `InternalMmapVector` из `sanitizer_common` оборачивает `mmap`-буфер в объект C++ со штатным деструктором, регистрируемым через `__cxa_atexit`. Освобождение этого буфера происходило бы при выходе из `main` — то есть до вызовов глобальных деструкторов прикладного кода, которые сами инструментированы и обращаются к стеку активных узлов. К моменту их исполнения страницы стека оказались бы удалены из адресного пространства, и любая попытка `push`/`pop` приводила бы к аварийному завершению. В реализации используется разновидность того же примитива — `InternalMmapVectorNoCtor` с отключёнными конструктором и деструктором: буфер остаётся жив всё время существования процесса и освобождается ядром при `_exit`.
6. `__seqsan_end` **после** `noreturn`**.** Если последний оператор в собранной хуком последовательности не может передать управление дальше (например, заканчивается на вызов функции с атрибутом `noreturn` вроде `exit` или `abort`), закрывающий вызов `__seqsan_end` оказывается заведомо недостижимым. В рамках `case`-ветки `switch` такой мёртвый вызов после `exit` становится для GCC дополнительным «живым» оператором между `exit` и следующим `case`, отчего срабатывает предупреждение `-Wimplicit-fallthrough`. Хук определяет подобную ситуацию через предикат `gimple_stmt_may_fallthru` над последним содержательным оператором последовательности и в этом случае закрывающего вызова не вставляет вовсе.

Все перечисленные в разделе 4.4 ситуации зафиксированы в виде регрессионных тестов в составе санитайзера (подраздел 4.5.1).

4.5 Тестирование

Тестирование санитайзера ведётся на двух уровнях. Первый — регрессионный набор внутри самого компилятора, фиксирующий построчно каждое реализационное решение из разделов

3 и 4 на минимальных целевых примерах. Второй — эксперимент по сборке и исполнению тестов подмножества пакетов дистрибутива Alpine Linux, выступающего крупным корпусом реального Си-кода: значительная часть пакетов снабжена авторскими тестами в рамках сборочной фазы `check`, что даёт готовые сценарии исполнения собранного кода.

4.5.1 Регрессионные тесты

В составе санитайзера собран регрессионный набор из 70 тестов, исполняемых под штатным для GCC механизмом DejaGnu: каждый тест — небольшая Си-программа, компилируемая с опцией `-fsanitize=sequence`, с проверкой ожидаемого наличия или отсутствия срабатывания. Покрытие подобрано так, чтобы каждое реализационное решение из подразделов 4.2–4.4 было закреплено хотя бы одним прямым тестом: все типы узлов модели и их сочетания, работа теневой памяти, оба режима поиска конфликтов (2 и 3) и все крайние случаи раздела 4.4.

4.5.2 Эксперимент на дистрибутиве Alpine Linux

Регрессионный набор подтверждает, что реализация соответствует описанной в работе модели на минимальных целевых примерах. Чтобы подтвердить применимость модели на реальном коде, помимо регрессионных тестов проведён эксперимент на репрезентативной выборке Си-проектов — по сборке и исполнению тестов пакетов дистрибутива Alpine Linux версии 3.23. Выбор Alpine обусловлен наличием готовой инфраструктуры массовой сборки его пакетов модифицированными версиями компилятора. Значительная доля пакетов дистрибутива снабжена авторскими тестами, исполняемыми в рамках сборочной фазы `check`.

Дистрибутив Alpine 3.23 содержит 9107 пакетов. Тестовая инфраструктура предварительно исключает из рассмотрения часть пакетов, заведомо неприменимых для тестирования модифицированного компилятора. Эта фильтрация значительно сокращает тестируемый корпус, однако не является полной. В предварительно исключённые пакеты входят:

1. 3327 пакетов, сборка которых не задействует компилятор Си — например, состоящих из сценариев на интерпретируемых языках, заголовочных файлов или документации;
2. 955 пакетов, сборочный сценарий которых задаёт собственные значения `CFLAGS`, перекрывая опции окружения, в результате чего опция `-fsanitize=sequence`, передаваемая через переменные окружения, до компилятора не доходит;
3. 279 пакетов, сборка которых завершается с ошибкой на любой версии GCC независимо от санитайзера;
4. 9 пакетов, передающих `CFLAGS` компилятору, отличному от GCC;
5. 6 пакетов, содержащих известные ошибки непосредственно в сборочном сценарии APKBUILD;
6. 5 пакетов с единичными техническими особенностями, не допускающими воспроизводимой сборки.

Совокупный объём перечисленных классов составляет 4581 пакет. Оставшиеся 4526 пакетов образуют *активный корпус* эксперимента. Каждый пакет активного корпуса собирался инструментированным компилятором — опция `-fsanitize=sequence` добавлялась во все стандартные переменные окружения сборки (`CFLAGS`, `CXXFLAGS`, `CPPFLAGS`, `LDFLAGS`), — после чего запускалась фаза `check`. Каждому пакету при сборке предоставлялось 8 потоков. Эксперимент проведён в двух режимах работы санитайзера: *WW+WR* (алгоритм 2) и *RW* (алгоритм 3). Переключение между режимами выполняется параметром `mode` переменной окружения `SEQSAN_OPTIONS` (подраздел 4.3.2).

4.5.3 Результаты эксперимента

Разбиение пакетов активного корпуса по категориям выполнено на основе результатов нескольких итераций тестирования каждого пакета в разных режимах работы санитайзера. Распределение пакетов активного корпуса по исходам приведено на рис. 19.

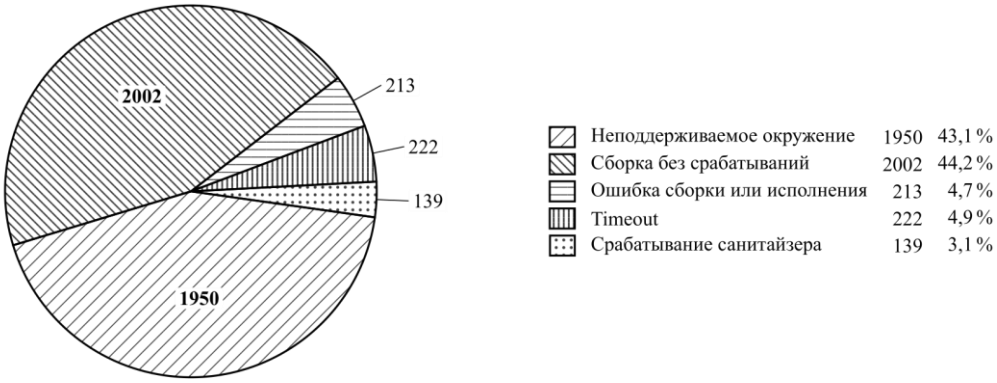

Рис. 19. Распределение пакетов активного корпуса по исходам эксперимента на дистрибутиве Alpine 3.23

Разберём каждый класс.

Сборка без срабатываний (2002 пакета, 44,2 %) образует основной класс, по которому можно судить о совместимости инструмента с реальным кодом.

Срабатывание санитайзера (139 пакетов, 3,1 %) объединяет пакеты, в которых `libseqsan` обнаружил хотя бы один *WW*/*WR*- или *RW*-конфликт. Эксперимент проводился с параметром `halt_on_error=0`, поэтому обнаружение конфликта не прерывает сборку. Диагностики санитайзера накапливаются в файле.

Неподдерживаемое окружение (1950 пакетов, 43,1 %) объединяет пакеты, в которых `-fsanitize=sequence` применяется к неподдерживаемому санитайзером фронтенду GCC либо к Си-коду в окружении, не подключающем `libseqsan`.

Ошибка сборки или исполнения (213 пакетов, 4,7 %) — пакеты, несовместимые с инструментацией санитайзера: например, опирающиеся на неподдерживаемые им механизмы вроде `ucontext`. Кроме того, инструментация может нарушать тесты, чувствительные к времени, объёму используемой памяти, количеству открытых файловых дескрипторов и прочему.

Timeout (222 пакета, 4,9 %) — пакеты, совокупное время на сборку и тестирование которых превысило установленный предел в 15 минут.

Подавляющее большинство из 139 срабатываний санитайзера имеет вид листинга 3 из подраздела 1.1.3: несколько вызовов аллокатора в аргументах одной функции. Такие конструкции не являются неопределённым поведением, но могут нарушать воспроизводимость вычислений от сборки к сборке. Настоящих случаев неопределённого поведения в корпусе обнаружено не было: репрезентативная выборка реального Си-кода, прошедшая через инструмент, оказалась свободна от unsequenced-модификаций. Конкретные примеры срабатываний из корпуса разобраны ниже.

4.5.4 Примеры срабатываний

Отчёт `libseqsan` сообщает LCA конфликтующих узлов, оба конфликтующих доступа с трассой стека и стек активных узлов, у каждого из которых указан класс (`S`, `EB` или `U` — см.

подразделы 4.2.2, 4.2.1) и текст подвыражения. Ниже приведены два характерных примера из корпуса; служебные кадры стека опущены, длинные пути сокращены многоточием.

Аллокация в аргументах одной функции (Apache APR)

В библиотеке Apache Portable Runtime 1.7.6 (`random/unix/apr_random.c`) функция `apr_random_standard_new` вызывает `apr_random_init`, три аргумента которого создают контексты SHA-256:

```
apr_random_init(r, p, apr_crypto_sha256_new(p),
                apr_crypto_sha256_new(p),
                apr_crypto_sha256_new(p));
```

Листинг 10. Три аллокации в аргументах одного вызова (Apache APR 1.7.6)

Три вызова `apr_crypto_sha256_new(p)` в аргументах одного `apr_random_init` не разделены точкой следования. Каждый из них через `apr_palloc(p, …)` изменяет служебные поля одного пула `p`, а порядок трёх инициализаций определяет компилятор:

```
Seqsan detected unsequenced access conflict at node 26398648
    (apr_random_init (r, p, apr_crypto_sha256_new (p), ...)):
  Access A: STORE (side-effect), Node 26398676
  Access B: LOAD, Node 26398693
  ...
  #  12 [U] Node 26398648, Info: apr_random_init (r, p, ...)
  #  13 [S] Node 26398683, Info: apr_crypto_sha256_new (p)
  ...
  #  17 [EB] Node 26398693, Info: if (... active->first_avail ...)
```

Наименьший общий предок — узел класса `U`, вызов `apr_random_init` с неупорядоченными аргументами; оба конфликтующих доступа — обращения к служебному полю одного пула из двух разных вызовов `apr_palloc`. Это неопределённо-упорядоченные аллокации (подраздел 1.1.3): неопределённого поведения нет, однако расположение аллокаций зависит от выбранного компилятором порядка. К этой категории относится большинство срабатываний санитайзера.

Ложное срабатывание на счётчике ссылок (jq)

Пример ложного срабатывания. В процессоре JSON jq 1.8.1 (`src/jq_test.c`) значения `jv` снабжены подсчётом ссылок, а `jv_copy` увеличивает счётчик переданного значения. Оба вызова `jv_copy` в аргументах одного `jv_equal` применены к одной переменной `a`:

```
assert(jv_equal(jv_copy(a), jv_copy(a)));
```

Листинг 11. Два инкремента счётчика ссылок в аргументах одного вызова (jq 1.8.1)

Санитайзер фиксирует конфликт инкрементов одного и того же счётчика `c->count` в теле `jv_copy`. Поскольку это один и тот же инкремент, любой порядок выполнения приводит к одинаковому результату — срабатывание ложное, того класса, что отмечен в разделе 3.1 (корректные программы на коммутативных операциях).

4.5.5 Накладные расходы

Безусловная инструментация каждого доступа к памяти, выбранная в подразделе 4.2.3, влечёт значительное замедление исполнения. Для оценки замедления была собрана выборка из семи Си-проектов с авторскими тестами — `zlib`, `xxhash`, `pcre2`, `expat`, `jansson`, `sqlite` и `libsodium`. Каждый проект собирался с одной из опций `-fsanitize=`, после чего на штатном тестовом наборе замерялось время исполнения относительно неинструментированной сборки. Результаты приведены на рис. 20.

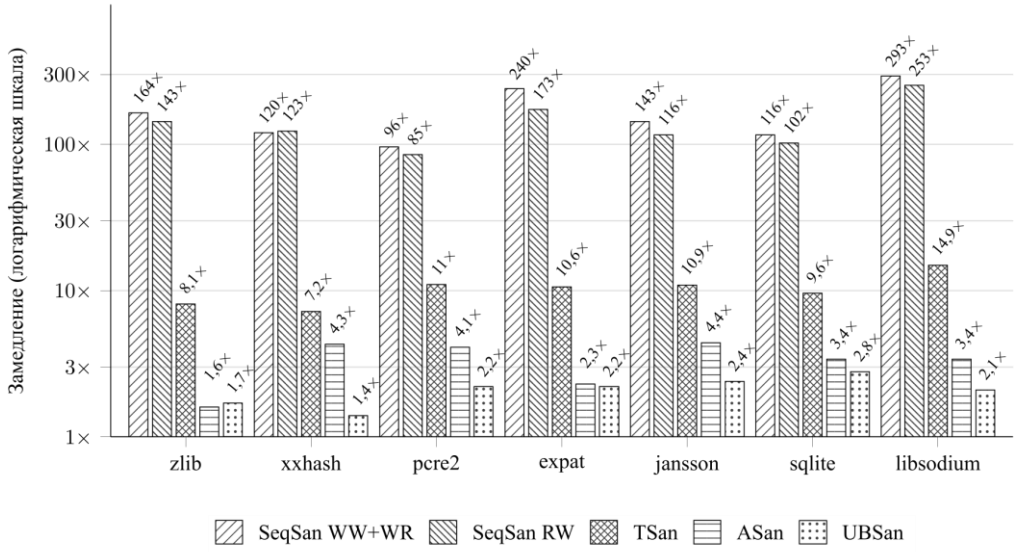

Рис. 20. Замедление исполнения для двух режимов работы и штатных санитайзеров GCC

Замедление, вызванное санитайзером, на выборке лежит в диапазоне 85–293×. ASan и UBSan замедляют исполнение в 1,4–4,4×, TSan — в 7,2–14,9×. Различие объясняется природой инструментации: санитайзер безусловно вызывает обработчик на каждом доступе к памяти, тогда как штатные санитайзеры GCC применяют статические оптимизации сокращения инструментации, исключающие проверки тех доступов, корректность которых может быть доказана на этапе компиляции. Снижение накладных расходов остаётся направлением дальнейшей работы. Возможны, в частности, следующие оптимизации:

1. сокращение числа порождаемых узлов модели. Вызов функции, а также условный и прочие операторы разворачиваются в несколько вложенных узлов. Часть этой инструментации в ряде случаев можно сократить без потери полноты;
2. отключение инструментации доступов к переменным, корректность использования которых доказуема статически — по образцу ASan-оптимизаций, отключённых в текущей реализации (подраздел 4.2.3);
3. исключение повторной инструментации доступов одного типа к одной и той же ячейке памяти в пределах одного узла.

Отдельно проверено предположение раздела 3.1 о том, что высота подъёма при поиске наименьшего общего предка (утверждение 2) мала и не зависит от объёма уже выполненных обращений к памяти. Для каждого проекта выборки на всех проверках упорядоченности фиксировалась высота подъёма по стеку активных узлов. Накопленные распределения этой величины для обоих режимов работы санитайзера приведены на рис. 21, сводные перцентили — в таблице 2.

Подъёмов за время работы накапливаются миллиарды, однако каждый из них обходится дёшево: половина проверок укладывается в четыре уровня, 99 % в режиме $WW{+}WR$ — в 32 уровня, в режиме RW — в 14. Поскольку реальные подъёмы коротки, линейный проход по стеку оказывается достаточным, и переход к более сложным алгоритмам поиска LCA не оправдан.

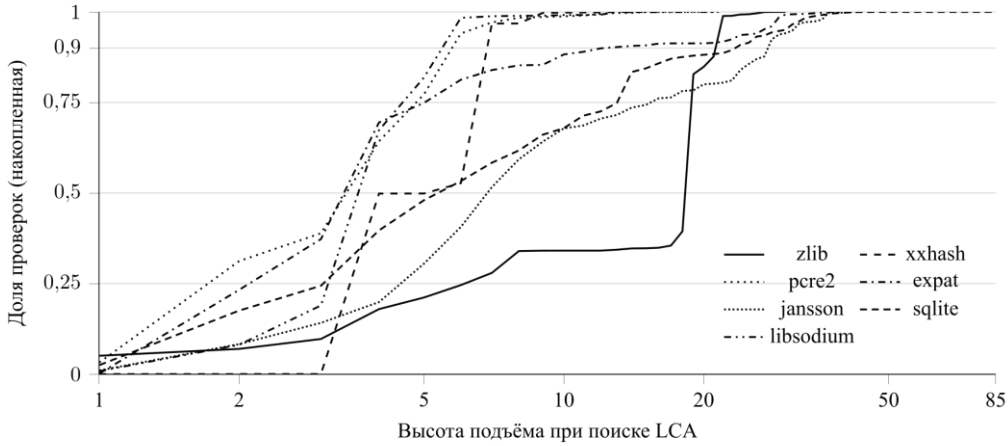

Рис. 21. Накопленное распределение высоты подъёма при поиске наименьшего общего предка в режимах WW+WR (а) и RW (б) Табл. 2. Перцентили высоты подъёма при поиске наименьшего общего предка (по всем проектам выборки)

Перцентиль	*WW+WR*	*RW*
50 %	4	4
75 %	7	5
90 %	19	8
99 %	32	14
Всего подъёмов	16,0 млрд	12,4 млрд

5. Заключение

Основным результатом настоящей работы является модель, позволяющая анализировать зависимости от порядка вычислений в программах на языке Си — иерархическая формализация отношения sequenced-before стандарта C11, допускающая эффективную динамическую проверку. Её применимость подтверждена реализацией санитайзера неупорядоченных модификаций на основе компилятора GCC и библиотеки `libsanitizer`.

Модель представляет исполнение программы деревом подвыражений с разметкой узлов точками следования и побочными эффектами (раздел 3.1). На нём сформулирован критерий упорядоченности через наименьшего общего предка пары узлов с доказанной корректностью для упрощённой (утверждение 1) и полной (утверждение 7) моделей, предложен алгоритм поиска LCA по стеку активных узлов за время, не зависящее от числа уже выполненных доступов (утверждение 2), и два алгоритма поиска конфликтов — *WW*/*WR* и *RW* (2, 3) — с постоянным объёмом теневой памяти на ячейку и доказанной совместной полнотой (утверждение 5).

Модель реализована как расширение GCC, активируемое опцией `-fsanitize=sequence` (раздел 4): новый проход `pass_seqsan` над GIMPLE и библиотека времени исполнения `libseqsan` в составе `libsanitizer` с теневой памятью по регионам адресного пространства Linux/x86_64. Реализована обработка крайних случаев языка Си и GCC (раздел 4.4), закреплённая 70 регрессионными тестами. Работоспособность продемонстрирована экспериментом на дистрибутиве Alpine Linux 3.23: из 4526 пакетов активного корпуса санитайзер сработал на 139 (3,1 %), а 2002 (44,2 %) собрались и прошли тесты без срабатываний. Подавляющее большинство срабатываний — неопределённо-упорядоченные аллокации в аргументах одного вызова, не являющиеся неопределённым поведением;

случаев собственно неопределённого поведения по C11 (§6.5, параграф 2) в корпусе не обнаружено. Замедление исполнения лежит в диапазоне 85–293× — следствие безусловной инструментации каждого доступа к памяти без статических оптимизаций, нацеленных на её сокращение.

Инструмент преодолевает фундаментальное ограничение статических анализаторов и при этом точно следует модели sequenced-before стандарта C11, не ограничивая число одновременно отслеживаемых переменных.

Направления дальнейшей работы включают:

1. снижение накладных расходов через статические оптимизации сокращения инструментации;
2. перенос реализации на другие фронтенды GCC и целевые платформы.

Список литературы

[1]. MITRE Corporation. CVE-2009-1897: Linux kernel TUN/TAP NULL pointer dereference. 2009. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897.
[2]. Nethercote Nicholas, Seward Julian. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. 2007. URL: https://valgrind.org/.
[3]. LLVM Project. AddressSanitizer — Clang documentation. 2024. URL: https://clang.llvm.org/docs/AddressSanitizer.html.
[4]. LLVM Project. UndefinedBehaviorSanitizer — Clang documentation. 2024. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.
[5]. International Organization for Standardization. ISO/IEC 9899:2011. Programming Languages — C. 2011. URL: https://www.iso.org/standard/57853.html.
[6]. International Organization for Standardization. ISO/IEC 9899:1999. Programming Languages — C. 1999. URL: https://www.iso.org/standard/29237.html.
[7]. International Organization for Standardization. ISO/IEC 14882:2017. Programming Languages — C++. 2017. URL: https://www.iso.org/standard/68564.html.
[8]. Free Software Foundation. Using the GNU Compiler Collection (GCC). 2024. URL: https://gcc.gnu.org/onlinedocs/gcc/.
[9]. Институт системного программирования им. В. П. Иванникова РАН. Svace: статический анализатор исходного кода. 2024. URL: https://www.ispras.ru/technologies/svace/.
[10]. Климов А.Ю. Автоматическое обнаружение гонок при параллельной сборке с использованием утилиты Make. Доклад на конференции OS DAY 2024, ИСП РАН. URL: https://osday.ru/2024/klimov.html.
[11]. Linux kernel contributors. Complete virtual memory map of 64-bit x86. 2024. URL: https://www.kernel.org/doc/html/latest/arch/x86/x86_64/mm.html.
[12]. Institute of Electrical and Electronics Engineers. IEEE Std 1003.1-2001/Cor 2-2004 (POSIX.1-2001 Technical Corrigendum 2). Standard for Information Technology — Portable Operating System Interface (POSIX). 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/functions/swapcontext.html.
[13]. Institute of Electrical and Electronics Engineers. IEEE Std 1003.1-2008 (POSIX.1-2008). Standard for Information Technology — Portable Operating System Interface (POSIX). 2008. URL: https://pubs.opengroup.org/onlinepubs/9699919799/.

Информация об авторах / Information about authors

Артем Юрьевич КЛИМОВ — магистрант Московского физико-технического института. Научные интересы: компиляторы, динамический анализ программ, санитайзеры, неопределённое поведение.

Artem Yurievich KLIMOV — master's student at the Moscow Institute of Physics and Technology. Research interests: compilers, dynamic program analysis, sanitizers, undefined behavior.

Александр Владимирович МОНАКОВ — старший научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: компиляторные технологии, оптимизация программ.

Alexander Vladimirovich MONAKOV — senior researcher in the Compiler Technology Department at ISP RAS. Research interests: compiler technologies, program optimization.

Владислав Анатольевич ИВАНИШИН — научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы включают в себя компиляторные технологии, безопасность программного обеспечения, методы статического и динамического анализа программ.

Vladislav Anatolievich IVANISHIN — researcher in the Compiler Technology Department at ISP RAS. Research interests include compiler technologies, software security, and methods of static and dynamic program analysis.

Дмитрий Михайлович МЕЛЬНИК — старший научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: компиляторные оптимизации, динамическая (JIT) компиляция.

Dmitry Mikhailovich MELNIK — senior researcher in the Compiler Technology Department at ISP RAS. Research interests: compiler optimizations, dynamic (JIT) compilation.