Федеральное государственное бюджетное учреждение науки
ИНСТИТУТ СИСТЕМ ИНФОРМАТИКИ им. А.П. ЕРШОВА
Сибирского отделения Российской академии наук

На правах рукописи

УДК 004.052.42

Кондратьев Дмитрий Александрович

МЕТОДЫ КОМПЛЕКСНОГО ПОДХОДА К АВТОМАТИЗАЦИИ ДЕДУКТИВНОЙ ВЕРИФИКАЦИИ ПРОГРАММ С ФИНИТНЫМИ ИТЕРАЦИЯМИ

05.13.11 – Математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель
к. ф.-м. н.
Промский Алексей Владимирович

Новосибирск – 2022

Оглавление

Введение 8

Глава 1. Методы дедуктивной верификации программ 22

1.1. Метод Флойда-Хоара дедуктивной верификации 22

1.1.1. Логика Хоара 22

1.1.2. Метод слабейшего предусловия 23

1.2. Метод Мориконі и Шварца метагенерации условий корректности 24

1.2.1. Формы правил вывода условий корректности 25

1.2.2. Построение генератора условий корректности по правилам вывода в нормальной форме 27

1.3. Метод Денни и Фишера анализа условий корректности с помощью семантических меток 28

1.3.1. Типы семантических меток 28

1.3.2. Разметка правил вывода семантическими метками 30

1.3.3. Алгоритм генерации текста о соответствии конструкций программ и подформул условий корректности 30

1.3.4. Пример объяснения условия корректности на естественном языке 31

1.4. Символический метод верификации финитных итераций 33

1.4.1. Финитные итерации 33

1.4.2. Представление финитных итераций с помощью операций замены 34

1.4.3. Генерация операции замены для финитных итераций без вложенных условных инструкций над неизменяемыми массивами 35

1.5. Языки C-light и C-kernel 37

1.5.1. Трансляция из C-light в C-kernel 38

1.5.2. Смешанная аксиоматическая семантика языка C-kernel . 41

1.6. Модуль трансляции из C-light в C-kernel 44

1.7. Модуль метагенерации условий корректности 47
1.7.1. Язык для задания шаблонов 47
1.7.2. Реализация метагенератора 48
1.7.3. Сопоставление программных конструкций и шаблонов . . 49
1.8. Модуль доказательства условий корректности 50
1.8.1. Критерии выбора системы доказательства 50
1.8.2. Входной язык системы ACL2 53
1.8.3. Система доказательства ACL2 57
1.9. Модуль трансляции конструкций C-kernel в конструкции C-light 59
1.10. Исходная версия системы C-lightVer 61
1.11. Выводы 63

Глава 2. Генерация условий корректности программ с финитными итерациями и стратегии автоматизации их доказательства 66
2.1. Алгоритм генерации функций, выражающих результаты итераций с вложенными условными инструкциями над неизменяемыми массивами 66
2.2. Алгоритм генерации функций на языке системы ACL2, выражающих результаты итераций над изменяемыми массивами языка C-kernel 67
2.2.1. Класс итераций, для которых применяется алгоритм генерации функций, выражающих результаты итераций над изменяемыми массивами 68
2.2.2. Трансляция тела цикла на язык системы ACL2 69
2.2.3. Генерация функций, выражающих результат обратных финитных итераций 77
2.2.4. Расширение языка шаблонов и задание на нем правила вывода для финитных итераций 78
2.3. Вспомогательные стратегии 79
2.3.1. Стратегия выбора посылок 79

2.3.2. Стратегия для программ, спецификации которых содержат функции со свойством конкатенации 82
2.3.3. Стратегия для программ с финитными итерациями над массивами 84
2.4. Основные стратегии 85
2.4.1. Стратегия интерактивного доказательства 85
2.4.2. Стратегия усиления условий корректности 88
2.4.3. Теорема о корректности стратегии усиления условий корректности и доказательство данной теоремы 94
2.4.4. Стратегия для программ, постусловием которых является разбор случаев выхода из цикла 99
2.5. Стратегии для классов финитных итераций 100
2.5.1. Стратегия для финитных итераций с инструкцией `break` 101
2.5.2. Стратегия для программ с выходом из цикла 102
2.5.3. Стратегия для финитных итераций над изменяемыми массивами 104
2.5.4. Стратегия для программ с вложенными циклами 105
2.6. Выводы 105

Глава 3. Метод автоматизации локализации ошибок 107
3.1. Стратегии локализации ошибок 107
3.1.1. Стратегия проверки ложности недоказанных условий корректности 107
3.1.2. Стратегия поиска циклов с неиспользуемыми присваиваниями элементам массива 110
3.1.3. Стратегия проверки исполнения инструкции `break` на первой итерации цикла 111
3.2. Генерация текстов о сопоставлении конструкций программы и подформул условий корректности 112
3.2.1. Язык задания текстовых шаблонов для типов семантических меток 112

3.2.2. Расширение языка представления правил вывода конструкцией семантических меток 113
3.2.3. Семантические метки для функций, выражающих результаты финитных итераций 115
3.2.4. Алгоритм генерации функций, выражающих результаты финитных итераций, с семантическими метками для итераций над изменяемыми массивами 116
3.2.5. Алгоритм генерации объяснений недоказанных условий корректности, содержащих операцию замены 120
3.3. Выводы 123

Глава 4. Применение разработанных методов для дедуктивной верификации C-программ с финитными итерациями 125
4.1. Модифицированная версия системы C-lightVer 125
4.2. Верификация программ с финитными итерациями над неизменяемыми массивами без инструкции `break` 128
4.2.1. Сумма абсолютных значений элементов вектора 128
4.2.2. Скалярное произведение векторов 130
4.3. Верификация программ с финитными итерациями над неизменяемыми массивами с инструкциями `break` 133
4.3.1. Проверка наличия в массиве не менее заданного количества вхождений ключа 134
4.3.2. Проверка наличия в массиве элемента, большего или равного заданному ключу 138
4.4. Программы с финитными итерациями над изменяемыми массивами с инструкциями `break` 145
4.4.1. Изменение знака первого отрицательного элемента массива на противоположный 145
4.4.2. Сортировка простыми вставками с инвариантом внешнего цикла и без инварианта внутреннего цикла 148
4.4.3. Сортировка простыми вставками без инвариантов циклов 156

4.5. Выводы 167

Глава 5. Применение разработанных методов для дедуктивной верификации Cloud Sisal программ 169

5.1. Система облачного параллельного программирования CPPS . . . 169

5.1.1. Язык Cloud Sisal 170

5.2. Язык Cloud-Sisal-kernel 173

5.2.1. Трансляционная семантика языка Cloud-Sisal-kernel . . . 174

5.2.2. Трансляция редукций на язык системы ACL2 175

5.2.3. Трансляция базовых выражений Cloud Sisal на язык системы ACL2 177

5.2.4. Алгоритм генерации структуры, хранящей переменные контекста 178

5.2.5. Алгоритм генерации функции *rep* 179

5.2.6. Алгоритм генерации функции *update_elements_id* . . . 181

5.2.7. Трансляция циклических выражений Cloud-Sisal-kernel на язык системы ACL2 182

5.2.8. Аксиоматическая семантика языка Cloud-Sisal-kernel . . . 182

5.3. Модули системы CPPS для дедуктивной верификации Cloud Sisal программ 183

5.4. Расширение языка C конструкциями языка Sisal, позволяющее применять в системе CPPS методы комплексного подхода системы C-lightVer 187

5.4.1. Синтаксис триплетов, циклических выражений, циклических выражений с условиями завершения и выражений замещения элементов массивов в языках C-light и C-kernel 188

5.4.2. Семантика триплетов, циклических выражений, циклических выражений с условиями завершения и выражений замещения элементов массивов в языке C-Sisal-kernel . . 190

5.5. Эксперименты по верификации программ на языках Cloud Sisal, Cloud-Sisal-kernel и C-Sisal-kernel 195

5.5.1. С-представление Sisal программы, проверяющей наличие в массиве не менее заданного количества вхождений ключа 196
5.5.2. С-представление Sisal программы, проверяющей упорядоченность элементов массива по возрастанию 204
5.5.3. Время исполнения реализации модуля CSV1 207
5.5.4. Сумма элементов матрицы на языке Cloud-Sisal-kernel . . 208
5.5.5. Произведение элементов матрицы на языке C-Sisal-kernel 210
5.5.6. Программа на языке C-Sisal-kernel, изменяющая знак первого отрицательного элемента массива на противоположный 212
5.6. Выводы 215

Заключение 216

Список литературы 219

Список сокращений и условных обозначений 238

Введение

Актуальность темы исследования. Автоматизация формальной верификации программ — актуальная задача современного программирования [44, 45, 50, 53, 68, 80, 87, 92, 93, 96, 97, 131, 132, 134, 152, 181]. Дедуктивная верификация [3, 27, 32, 37, 38, 82, 85, 93, 95] является сопоставлением программы и ее спецификаций, заданных в виде логических формул. Типичными спецификациями являются пред- и постусловие программы, а также инварианты циклов. Классический процесс верификации заключается в выводе условий корректности (УК) с помощью генератора условий корректности (ГУК), который воплощает в себе аксиоматическую семантику целевого языка программирования. Далее УК проверяются с помощью системы поддержки доказательства теорем. В случае истинности всех УК пара программа-спецификации является согласованной.

Примером подобной системы верификации является проект C-lightVer, реализуемый в Лаборатории теоретического программирования ИСИ СО РАН [147, 148]. Входной язык C-light [28, 31, 33, 156] — выразительное подмножество стандарта C99 [164]. В языке C-light выделено ядро — язык C-kernel, для которого была разработана аксиоматическая семантика в стиле Хоара [29, 31, 33, 157]. Процесс дедуктивной верификации [3, 27, 32, 37, 38, 82, 85, 93, 95] разбивается на три этапа. На первом этапе аннотированная C-light программа транслируется в эквивалентную аннотированную программу на языке C-kernel [30, 31, 157]. Далее с помощью аксиоматической семантики языка C-kernel выводятся условия корректности (УК). Потом следует этап доказательства полученных УК.

Отметим, что ранее в системе C-lightVer была реализована процедура, позволяющая от конструкций промежуточной C-kernel программы вернуться к конструкциям исходной C-light программы [148], а также метод смешанной аксиоматической семантики [148] и метод метагенерации УК [124, 151]. Процедура сопоставления промежуточной C-kernel программы и исходной C-light программы позволяет упростить нахождение ошибки в исходной программе в случае ее нахождения в промежуточном представлении [8]. Метод смешанной аксиоматической семантики позволяет использовать специальные версии пра-

вил вывода для определенных программных конструкций, что может приводить к генерации более простых УК [26, 148]. Метагенерация УК позволяет использовать правила вывода УК как входные данные системы верификации, что может упростить задачу расширения генератора УК новыми правилами вывода в случае расширения входного языка новыми конструкциями [21].

Исследования и разработки, осуществлявшиеся автором в проекте C-lightVer в период 2014–2022 гг. послужили основным научным и практическим заделом данной диссертации.

Несмотря на более чем 50-летний период развития теории дедуктивной верификации, в ней можно выделить ряд ключевых проблем, из-за которых она по-прежнему остается во многом уделом академической среды и непопулярна среди обычных программистов. Основной целью диссертации стала разработка комплекса методов, направленных на решение трех проблем: проблемы инвариантов циклов, проблемы локализации и объяснения ошибок и проблемы автоматизации доказательства УК.

Рассмотрим проблему инвариантов циклов [86]. Инвариант цикла — это утверждение, которое должно выполняться перед началом исполнения, на любой итерации и по завершению цикла. В общем случае задача автоматической генерации инвариантов циклов алгоритмически неразрешима [32], поэтому частичным решением может стать выделение класса алгоритмов, для которых можно вообще обойтись без инвариантов. За основу в диссертации был взят символический метод верификации финитных итераций [158]. Он применяется к циклам специального вида (финитным итерациям). Тело финитной итерации исполняется один раз для каждого элемента структуры данных конечной размерности. Несмотря на кажущуюся вырожденность, эта ситуация в точности соответствует такому фундаментальному разделу информатики, как итеративная обработка структур данных последовательной природы — строки, массивы, списки, очереди, частично деревья. Основой этого метода является символическая замена финитных итераций специальными рекурсивными функциями, называемыми операциями замены. В рамках проекта РФФИ № 17-01-00789 «Платформенно-независимый подход к формальной спецификации и верификации

стандартных математических функций» важной задачей являлась разработка алгоритма генерации операций замены [117], позволяющего генерировать условия корректности для программ с финитными итерациями без использования инвариантов циклов. Разработанный алгоритм стал одним из результатов диссертации.

Рассмотрим проблему локализации и объяснения ошибок. Исторически дедуктивная верификация была ориентирована не на локализацию ошибок, а на доказательство отсутствия ошибок [149]. Поэтому значительный интерес представляет задача интерпретации недоказанных УК и соотнесение их с местом потенциальной ошибки в программе или в спецификации.

Денни и Фишер предложили добавить в правила вывода УК семантическую разметку для объяснения результата применения правила [74]. Генератор УК в процессе вывода добавляет к различным подформулам соответствующие метки, которые извлекаются из УК и переводятся в текст о соответствии УК и фрагментов программы. Отчет о результатах верификации, основан на информации, хранящейся в семантических метках.

Но Денни и Фишер предложили лишь ограниченный набор типов семантических меток. В рамках проекта РФФИ № 11-01-00028 «Интегрированный мультиязыковый подход к верификации императивных программ» важной задачей являлась разработка языка, позволяющего использовать в правилах вывода предложенные пользователем семантические метки [18, 147]. Разработанный язык стал одним из результатов диссертации. Также результатом диссертации стала разработка семантических меток для финитных итераций [13, 19, 121, 126]. Для более точной локализации ошибок оказалось полезным проверять циклы на выполнение определенных ошибочных свойств и генерировать объяснения в случае выполнения таких свойств. На основе таких проверок был разработан ряд автоматизированных стратегий локализации ошибок [121, 126–128].

Рассмотрим проблему автоматизации доказательства УК. Несмотря на достаточную развитость систем поддержки доказательства теорем, они не всегда справляются с доказательством УК в полностью автоматическом режиме. Причиной тому может быть необходимость доказательства по индукции из-за на-

личия в УК применений рекурсивных функций, сложная структура УК и т.д. Поэтому, актуальна задача разработки стратегий для различных классов программ, позволяющих автоматизировать доказательство УК этих программ. В рамках проекта РФФИ № 15-01-05974 «Онтологический подход к формальной семантике языков программирования» важной задачей являлась разработка стратегий автоматизации доказательства УК, содержащих применение функций, выражающих результаты финитных итераций [117]. Разработанные стратегии стали одним из результатов диссертации.

Одним из результатов диссертации стал набор стратегий, позволяющих автоматизировать доказательство УК программ с финитными итерациями без использования инвариантов циклов [11, 14, 117–123, 125–128, 144]. Важным достоинством набора стратегий стало доказательство их корректности.

Рассмотрим расширение методов на другие языки и парадигмы и общее повышение автоматизируемости процесса верификации. Помимо разработки теоретических методов решения рассмотренных проблем, диссертация нацелена и на решение важных практических задач. Интерес представляют методы, пригодные не только для языка C, но и для других языков и парадигм программирования.

Важным результатом диссертации стало применение разработанных методов и стратегий в проекте по созданию Системы облачного параллельного программирования (CPPS) [2, 4, 104, 105, 107, 168]. Cloud Sisal [5, 6, 103, 105, 108, 176] является входным языком системы CPPS. Cloud Sisal является функциональным языком программирования, основанным на циклических выражениях. Главной особенностью системы CPPS является неявное параллельное исполнение, основанное на автоматическом распараллеливании циклов Cloud Sisal [2, 108, 168]. Кодогенератор системы CPPS позволяет генерировать код на языке C, соответствующий различным этапам оптимизации входной программы [2]. Работа в проекте шла по двум направлениям: применение системы C-lightVer к готовому промежуточному представлению Sisal-программы на языке C [122, 123, 125] и непосредственная разработка аксиоматической семантики Cloud Sisal [24, 130] с расширением системы C-lightVer новыми правилами

[15, 16, 106, 130]. В рамках проекта РНФ № 18-11-00118 «Облачные методы и средства конструирования эффективных и надежных параллельных программ на основе функциональных спецификаций и семантических преобразований» важной задачей являлось создание расширения языка C конструкциями Cloud Sisal [15, 16], семантики такого расширения [15, 16] и реализация такой семантики в системе C-lightVer [15]. Аксиоматическая семантика созданного расширения языка C конструкциями Cloud Sisal стала одним из результатов диссертации.

Итоговым результатом данной диссертации стал комплексный подход к автоматизации дедуктивной верификации без использования инвариантов циклов. Комплекс включает в себя метод, позволяющий генерировать УК для программ с финитными итерациями без инвариантов циклов, стратегии, позволяющие автоматизировать доказательство порождаемых УК и метод, позволяющий автоматизировать локализацию и объяснение ошибок для программ с финитными итерациями. Комплексный подход оказался применимым для языков C и Cloud Sisal, которые являются яркими представителями процедурной и функциональной парадигмы соответственно.

Степень разработанности темы исследования и обзор родственных работ. Рассмотрим работы об автоматизации дедуктивной верификации программ с циклами. Подход, похожий на символический метод верификации финитных итераций, предложили Майрен и Гордон (Myreen,Gordon) [154]. Они также предлагают заменять действие цикла на результат применения рекурсивной функции. Но Майрен и Гордон рассматривают модельный язык, более простой, чем язык C-light. Подобный подход также предложен в работе Бланка и др. (Blanc, etc.) [52] для Scala-программ. Однако, в работе [52] предлагается использовать инварианты, которые транслируются в аннотации соответствующих циклам рекурсивных функций. В отличие от символического метода верификации финитных итераций, большинство работ в этой области основано на генерации инвариантов циклов. Ковач (Kovács) [97, 134] предложила метод, основанный на базисах Гребнера, для генерации инвариантов P-разрешимых циклов. Но, в отличие от финитных итераций, правые части присваиваний в те-

ле Р-разрешимых циклов должны иметь вид полиномов. В работе Чакраборти и др. (Chakraborty, etc.) [58] предлагается генерировать инварианты специального вида для циклов над массивами. Но авторы не рассматривают циклы с инструкцией `break`. Будем называть свойство перестановочности результирующего массива по отношению к исходному массиву свойством перестановочности. В работе Галеотти и др. (Galeotti, etc.) [88] предложен динамический метод, основанный на известной идее модификации постусловия, но авторы не доказывали свойство перестановочности для различных программ сортировки. Отказ от доказательства этого свойства привел к генерации таких инвариантов циклов, которые могут позволить доказать только свойство упорядоченности. В работе Сриваставы и др. (Srivastava, etc.) [175] предложено использовать шаблоны инвариантов, задаваемые пользователями, однако, для классических программ сортировки доказано более слабое свойство, чем свойство перестановочности.

Рассмотрим работы в области автоматизации доказательства УК. Распространенным подходом [102] является генерация таких лемм, которые могут помочь доказать целевую теорему. Такой подход для системы доказательства ACL2 [98, 109, 110, 150] предложили Херас и др. (Heras, etc.) [94] В отличие от предложенных нами стратегий доказательства для системы ACL2, стратегии Хераса и др. основаны на машинном обучении. Но машинное обучение плохо подходит для доказательства УК, так как предметные области могут сильно отличаться для разных программ. В работе Имине и Ранизе (Imine, Ranise) [99] были предложены стратегии доказательства для УК сортировки вставками, однако не все УК были доказаны автоматически. Актуальность разработки стратегий автоматизации доказательства УК сортировки вставками подтверждает работа Жианга и Жоу (Jiang, Zhou) [101]. В работе Сафари и Хьюсман (Safari, Huisman) [172] предложены стратегии автоматизации доказательства свойства перестановочности для известных программ сортировки, но в этих экспериментах авторы задавали инварианты для каждого цикла. В работе де Анжелиса и др. (de Angelis, etc.) [70] на базе модельного функционального языка предложена стратегия для автоматизации верификации сортировки с помощью дизъюнктов Хорна. Дизъюнкты Хорна могут упростить доказательство свойств про-

грамм над рекурсивными типами данных [133], но при использовании такого подхода требуются специальные стратегии [180]. Для системы доказательства Lean были предложены тактики для автоматизации доказательства по индукции, основанные на преобразовании индукционных гипотез [143]. Применение этих тактик было продемонстрировано для доказательства свойств программ на модельном языке [143]. Лейно (Leino) [138] предложил способ автоматизации доказательства по индукции для SMT-решателя Z3 [51, 73]. Этот способ основан на классической идее доказательства шага индукции и доказательстве индукционного перехода. Однако, для автоматизации доказательства УК такой тактики недостаточно [144, 171]. Рейнольдс и Кунчак (Reynolds, Kuncak) [171] предложили набор стратегий автоматизации доказательства по индукции для SMT-решателя CVC4 [43]. Главной из этих стратегий является индукция по рекурсивному определению структуры данных. Подобная стратегия стала основной стратегией автоматизации доказательства по индукции в новой версии системы Vampire [170]. Но индукции по рекурсивному определению структуры данных также недостаточно для автоматизации доказательства УК [144].

Рассмотрим работы со специализированными стратегиями, ориентированными на упрощение формальной верификации. Туэрк (Tuerk) [178] предлагает задавать для циклов предусловия и постусловия специального вида вместо инвариантов. Обобщение данного подхода описано в статье Эрнста (Ernst) [79]. В работе Волкова и др. [181] рассматривается метод лемма-функций, реализованный в системе AstraVer. Этот метод основан на использовании спецификаций специального вида вместо инвариантов. В работе Бланчарда и др. (Blanchard, etc.) [53] описан похожий метод, реализованный в системе Frama-C [45]. Однако, эти методы основаны на задании спецификаций пользователем [53, 178, 181].

Рассмотрим работы об автоматизации локализации ошибок при дедуктивной верификации. Дайлер и др. (Dailler, etc.) [67] описали использование контрпримера, сгенерированного SMT-решателем, для локализации ошибок. Но анализ контрпримера может оказаться достаточно сложным, что продемонстрировано в работе Бекера и др. (Becker, etc.) [47]. Денни и Фишер (Denney, Fischer) [74] предложили способ установления соответствия между УК и исход-

ным кодом. В работе Кенигхофера и др. (Könighofer, etc.) [131] описан новый подход в системе Frama-C [45] к автоматизации локализации ошибок при дедуктивной верификации, основанный на изменении выражений программы. Раад и др. (Raad, etc.) предложили логику, которую они назвали некорректной логикой разделения [169]. Истинность специальных формул в этой логике означает наличие ошибок в программе. Однако, Раад и др. предложили сложную модель памяти, приводящую к генерации сложных формул в отличие от используемого нами метода смешанной аксиоматической семантики [147]. В работе де Гоува и др. (de Gouw, etc.) [72] описана локализация ошибки при верификации реализации сортировки в Java-машине OpenJDK. Ошибка была локализована с помощью доказательства невыполнения инварианта в определенных случаях. Но решение проблемы инвариантов циклов не было автоматизировано в работах [72, 131, 169]. В работе Меллера и др. (Möller, etc.) [149] предложен единый подход к доказательству корректности программ и наличия ошибок в программах. Комплексный подход, описанный в данной диссертации, также является единым подходом к этим проблемам. Однако, комплексный подход основан на доказательстве выполнения свойств функций, выражающих результаты финитных итераций, и, в отличие от подхода Моллера и др., содержит методы для решения проблемы инвариантов в случае определенных классов циклов.

Рассмотрим работы про промежуточные языки верификации программ. Наиболее [140] распространенными промежуточными языками верификации являются Boogie [35, 40, 141] и WhyML [56, 84, 132]. К примеру, следующие системы используют Boogie [179]: AutoProof [87, 177] для верификации Eiffel-программ [159, 163], VCC [59, 60, 152, 153] для верификации C-программ, Spec# [41, 42] для верификации C#-программ и система [137, 139] верификации Dafny-программ. WhyML (предоставляемый платформой Why3 [56, 84, 132]) используется в следующих системах: Frama-C [45] для анализа C-программ, AstraVer [78, 181] для верификации C-программ и SPARK 2014 [132] для верификации Ada-программ. Также платформа Why3 (и ее предыдущие версии [83]) используются системой Krakatoa [76, 83] для верификации Java-программ и утилитой WP [65, 66] для верификации C-программ в системе Frama-C. Но из-за проблем [57, 132] с

автоматизацией верификации Boogie-программ и WhyML-программ для систем верификации Dafny-программ и Ada-программ вместо прямой работы с промежуточными языками верификации были разработаны специальные скриптовые языки для высокоуровневого описания доказательства [68, 92]. В системе C-lightVer промежуточным языком верификации является C-kernel [157], подмножество входного языка C-light [156]. Преимуществом такого подхода является одинаковая операционная семантика входного и промежуточного языка. Это облегчает доказательство сохранения семантики при трансляции с входного языка в промежуточное представление [31].

Рассмотрим работы про задание формальных семантик языков программирования. Формальная семантика для современного стандарта C11 [165] была разработана Кребберсом и Виедиджком (Krebbers, Wiedijk) [135]. Саммлер и др. (Sammler, etc.) [173] разработали новую систему дедуктивной верификации C-программ, в которой семантика языка C задана в системе Coq [161]. В проекте VERISOFT [34, 89] по верификации ядра операционной системы используется язык C0 [136]. Семантика языка C0 моделируется в системе доказательства теорем Isabelle/HOL [162]. Отметим, что C0 полностью покрывается языком C-light. В проекте по созданию верифицированного компилятора CompCert [49, 142] для языка C в качестве входного и промежуточного языка используются следующие подмножества C: C_{light} [54] и C_{minor} [36]. Семантика конструкций функциональной парадигмы программирования в новейших стандартах C++ и Java описана в работах Кока и Тасирана (Cok, Tasiran) [63, 64], для Java такая семантика реализована в проекте OpenJML [61, 62]. В системе KeY [174] для верификации Java-программ используется семантика, называемая JavaDL [48, 91]. Недостатком всех вышеперечисленных семантик являются правила вывода для циклов и итераций, которые требуют задания инвариантов.

Рассмотрим работы про расширение языков программирования специальными видами циклов и их семантику. Ранее Аттали и др. (Attali, etc.) [39] разработали для циклических выражений Sisal натуральную семантику, но она больше подходит для разработки компиляторов, чем для дедуктивной верификации. Библиотека MapReduce [71] расширяет императивные и объектно-ориентиро-

ванные языки программирования конструкциями, подобными функциям *map* и *reduce* из функциональной парадигмы программирования. Операционная семантика для расширения языка C-like конструкциями OpenMP была предложена в работе Блома и др. (Blom, etc.) [55]. Новейший стандарт C++20 [166] вводит диапазоны (ranges) и итерации над ними. Более того, диапазоны (ranges) сходны с триплетами языка Cloud Sisal. Такая особенность Sisal, как конструкции, подобные `break`, является преимуществом Sisal-циклов относительно рассмотренных видов итераций. Актуальность автоматизации дедуктивной верификации в случае циклов с инструкциями `break` продемонстрирована примерами из набора задач по верификации, созданном Якобсом и др. (Jacobs, etc.) [100].

Цели и задачи диссертационной работы:

Целью научной работы является разработка методов, обеспечивающих автоматизацию, расширяемость и проблемную ориентированность комплексного похода к формальной верификации программ без использования инвариантов циклов. Для достижения данной цели были поставлены следующие задачи:

1. Разработка метода, позволяющего генерировать условия корректности для программ с финитными итерациями без использования инвариантов циклов.

2. Разработка стратегий, позволяющих автоматизировать доказательство условий корректности программ с финитными итерациями. Доказательство корректности данных стратегий.

3. Разработка метода, позволяющего автоматизировать локализацию ошибок при дедуктивной верификации программ с финитными итерациями.

4. Разработка аксиоматической семантики языка Cloud Sisal, позволяющей проводить дедуктивную верификацию программ на данном языке без использования инвариантов циклических выражений. Разработка аксиоматической семантики расширения языка C конструкциями языка Sisal, позволяющей применять в системе CPPS методы комплексного подхода системы C-lightVer.

5. Реализация методов комплексного подхода в системе C-lightVer, позволяющая при дедуктивной верификации программ с финитными итерациями, заданных на языках C, Cloud Sisal и на расширении C конструкциями Cloud Sisal, автоматизировать доказательство условий корректности и автоматизировать локализацию ошибок. Проведение экспериментов по автоматизированной верификации программ на данных языках.

Соответствие диссертации паспорту специальности. Цели и задачи исследования соответствуют следующим пунктам паспорта специальности 05.13.11: модели, методы и алгоритмы проектирования и анализа программ и программных систем, их эквивалентных преобразований, верификации и тестирования (пункт 1); языки программирования и системы программирования, семантика программ (пункт 2); программные системы символьных вычислений (пункт 5).

Методология и методы исследования. Методология исследования базируется на подходах информатики к формальной верификации программ. В работе используется метод дедуктивной верификации программ, метод слабейшего предусловия, символический метод верификации финитных итераций, метод метагенерации условий корректности для упрощения расширения генератора УК, метод смешанной аксиоматической семантики для упрощения УК, стратегии автоматизации доказательства УК, метод семантической разметки для локализации ошибок.

Положения, выносимые на защиту:

1. Разработанный метод генерации операции замены для циклов позволяет генерировать условия корректности для программ с финитными итерациями без использования инвариантов циклов.

2. Разработанные стратегии доказательства условий корректности позволяют автоматизировать проверку на истинность условий корректности программ с финитными итерациями.

3. Разработанный метод локализации ошибок позволяет автоматизировать

сопоставление конструкций программы и подформул условий корректности, а также локализацию ошибок в программах с финитными итерациями.

4. Разработанная аксиоматическая семантика языка Cloud-Sisal-kernel позволяет проводить дедуктивную верификацию программ на данном языке без использования инвариантов циклических выражений. Разработанная аксиоматическая семантика расширения (C-Sisal-kernel) языка C конструкциями языка Sisal позволяет применять в системе CPPS методы комплексного подхода системы C-lightVer.

5. Реализация методов комплексного подхода в системе C-lightVer позволяет при дедуктивной верификации программ с финитными итерациями, заданных на языках C, Cloud-Sisal-kernel и C-Sisal-kernel, автоматизировать доказательство условий корректности и автоматизировать локализацию ошибок. Были проведены эксперименты по автоматизированной верификации программ на данных языках.

Научная новизна. Научная новизна работы состоит в

1. Автоматизации доказательства УК программ с финитными итерациями без использования инвариантов циклов.

2. Автоматизации локализации ошибок при дедуктивной верификации программ с финитными итерациями без использования инвариантов циклов.

3. Применимости разработанного комплексного подхода к программам с финитными итерациями на языках императивного программирования (на примере языка C), на языках функционального программирования (на примере языка Cloud Sisal) и на языках, основанным на обеих парадигмах программирования (на примере языка C-Sisal-kernel).

Теоретическая и практическая значимость. Теоретическая значимость работы состоит в разработке методов для комплексного подхода [121], которые позволяют в случае финитных итераций решить проблемы инвариантов

циклов, автоматизации доказательства УК и автоматизации локализации ошибок. Данный подход может быть применен к широкому классу языков императивного и функционального программирования, позволяющих задавать финитные итерации над структурами данных. Это продемонстрировано применением комплексного подхода для автоматизации верификации программ на императивном языке C, программ на функциональном языке Cloud Sisal и программ на языке C-Sisal-kernel, основанным на обеих парадигмах программирования.

Практическая значимость работы состоит в реализации прототипа системы верификации C-lightVer [17, 121]. Данная система применима для верификации программ на языках, предствляющих две парадигмы программирования: язык C, язык Cloud Sisal и язык C-Sisal-kernel.

Степень достоверности и апробация результатов. Достоверность и обоснованность результатов исследования обеспечивается формальными доказательствами, а также компьютерными экспериментами. Основные результаты работы докладывались на следующих научных конференциях и семинарах: международные научно-практические конференции PSI-2017 [114], PSI-2019 [118], TOOLS-2019 [125], SIBIRCON-2019 [123], TMPA-2015 [18, 167], международные семинары PSSV-2017 [145], PSSV-2018 [120], PSSV-2019 [127], PSSV-2021 [15], всероссийские конференции молодых ученых по математическому моделированию в 2015 году [23], 2016 году [20], в 2017 году [10], в 2018 году [11], в 2019 году [14], в 2020 году [16] и в 2021 году [13], всероссийский онлайн-семинар ru-STEP (записи доступны на видеоканале ИСИ СО РАН [1]), научно-исследовательские семинары ИСИ СО РАН, коллоквиумы ИСИ СО РАН [7].

Результаты, полученные в ходе выполнения данной работы, применены на практике в следующих проектах: РФФИ № 11-01-00028-а, РФФИ № 15-01-05974, РФФИ № 17-01-00789 и РНФ № 18-11-00118.

Публикации. Материалы диссертации опубликованы в 38 [4, 9–16, 18–20, 22–25, 106, 114, 117–130, 144–148, 167] печатных работах, в том числе 14 [15, 19, 106, 114, 117, 118, 121, 123–126, 130, 144, 147] статей в изданиях перечня ВАК, из них 11 публикаций [106, 114, 117, 118, 121, 123–126, 130, 147], входящих в международные базы цитирования Scopus и Web of Science. Было получено

свидетельство о государственной регистрации программы для ЭВМ [17].

Положения, выносимые на защиту в данной диссертации, описаны в следующих публикациях: метод генерации операции замены описан в публикациях [11, 117, 121, 126–128, 144], стратегии доказательства условий корректности описаны в публикациях [14, 117–123, 125, 126, 144], метод локализации ошибок описан в публикациях [10, 13, 18–20, 114, 121, 126–128, 147, 148], семантика языка Cloud-Sisal-kernel и расширение ей языка C-kernel описаны в публикациях [4, 15, 16, 24, 130], реализация методов комплексного подхода для языков Cloud-Sisal-kernel и C-Sisal-kernel описана в публикациях [4, 15–17, 106, 122, 123, 125, 130], реализация методов комплексного подхода для языка C-light описана в публикациях [9–12, 14, 17–20, 22, 23, 25, 114, 117–129, 144, 145, 167].

Положения, выносимые на защиту, описанные в работах с несколькими соавторами [4, 24, 25, 106, 117–130, 144–148], получены лично автором.

Личный вклад автора. Содержание диссертации и основные положения, выносимые на защиту, отражают персональный вклад автора в опубликованные работы. Подготовка к публикации полученных результатов проводилась совместно с соавторами, причем вклад диссертанта был определяющим. Все представленные в диссертации результаты получены лично автором.

Структура и объем диссертации. Диссертация состоит из введения, 5 глав, заключения и библиографии. Общий объем диссертации 238 страниц, из них 212 страниц текста, включая 7 рисунков и 2 таблицы. Библиография включает 181 наименование на 18 страницах.

Глава 1

Методы дедуктивной верификации программ

В данной главе опишем методы, составляющие основу дедуктивной верификации (логика Хоара, метод слабейшего предусловия), и методы, используемые в комплексном подходе для решения проблем, возникающих при дедуктивной верификации программ (метод метагенерации условий корректности, метод семантической разметки, символический метод верификации финитных итераций). Также рассмотрим задел по системе дедуктивной верификации C-lightVer, созданный до работы над данной диссертацией, а также системы доказательства теорем, которые использовались ранее и используются сейчас в данной системе в качестве модуля доказательства.

1.1. Метод Флойда-Хоара дедуктивной верификации

Верификация программ в дедуктивных системах традиционно основана на генерации условий корректности (УК) [3, 27, 32, 37, 38, 82, 85, 93, 95]. Она заключается в формальном доказательстве корректности программ в соответствии с описанием их свойств, задаваемых в виде спецификаций программ.

1.1.1. Логика Хоара

В 1969 году Ч. Хоар [95] ввел способ задания аксиоматической семантики, ставший основой метода дедуктивной верификации программ. Подход Хоара [38] заключается в том, чтобы представлять текст программы как особое отношение между утверждениями. Базовыми формулами в рассматриваемом подходе являются тройки Хоара $\{P\}\ S\ \{Q\}$, где P — предусловие (логическая формула), S — программа, Q — постусловие (логическая формула). Частичная корректность тройки Хоара $\{P\}\ S\ \{Q\}$ означает, что "если предусловие P истинно перед исполнением фрагмента программы S, и, если исполнение S завершилось, тогда постусловие Q выполняется после его завершения" [151].

Правила вывода задаются в виде

$$\frac{\psi_1, ..., \psi_n}{\varphi}$$

где ψ_1,...,ψ_n – посылки правила вывода (набор троек Хоара и логических формул) и φ — заключение правила вывода (тройка Хоара). Данная нотация означает, что φ выводимо при гипотезе ψ_1,...,ψ_n. Семантика простых операторов (например, присваивания) задается, как правило, с помощью набора аксиом, а любого сложного оператора (например, оператора последовательного исполнения) – с помощью правила вывода. Логическая система, содержащая аксиомы и правила вывода для всех синтаксических форм языка программирования, называется логикой Хоара или аксиоматической семантикой языка.

В качестве примера рассмотрим классическое правило вывода для цикла `while`:

$$\frac{\{P\}\ \mathbf{prog};\ \{I\},\ \ \{I \wedge B\}\ \mathbf{S}\ \{I\},\ \ I \wedge \neg B \rightarrow Q}{\{P\}\ \mathbf{prog};\ \ \mathbf{while\ B\ inv\ I\ do\ S}\ \{Q\}}$$

Чтобы вывести тройку Хоара для цикла `while`, необходимо использовать индукцию. Ввести индукцию позволяет специальная логическая формула, приписываемая циклу и называющаяся инвариантом цикла. Инвариант цикла – это утверждение, которое истинно перед исполнением цикла и для каждой итерации цикла и обеспечивает корректность на выходе из цикла. Таким образом, посылками правила вывода для цикла с предусловием являются тройка Хоара, соответствующая входу в цикл, тройка Хоара, соответствующая итерации цикла без выхода из него, и логическая формула, соответствующая выходу из цикла.

1.1.2. Метод слабейшего предусловия

Генератор условий корректности (ГУК) осуществляет вывод в автоматическом режиме по аксиоматической семантике и сводит частичную корректность тройки Хоара $\{P\}\ S\ \{Q\}$ к истинности некоторого числа лемм, называемых

условиями корректности, в предметной области. Доказуемости этих лемм достаточно для частичной корректности исходной аннотированной программы.

Одним из методов такого вывода является метод слабейшего предусловия [151]. Для тройки Хоара $\{P\}\ S\ \{Q\}$ слабейшее предусловие обозначается как $wp(S, Q)$ [38]. Слабейшее предусловие для тройки Хоара – это такое предусловие, которое обеспечивает частичную корректность данной тройки Хоара, и оно выводимо как логическая формула из всех остальных предусловий, обеспечивающих истинность тройки Хоара. Значит, если $wp(S, Q)$ является слабейшим предусловием программы S с постусловием Q и P является формулой, тогда тройка $\{wp(S, Q)\}\ S\ \{Q\}$ частично корректна и из истинности формулы $P \to wp(S, Q)$ следует частичная корректность тройки $\{P\}\ S\ \{Q\}$. Данное свойство позволяет генерировать УК в виде $P \to wp(S, Q)$. Таким образом, ГУК может быть реализован как программа, вычисляющая $wp(S, Q)$.

1.2. Метод Морикони и Шварца метагенерации условий корректности

Когда верификацию изучают на примере простого модельного языка, то в такой период обычно не требуется вносить изменения в ГУК. Но при использовании дедуктивной верификации на практике [93] появляется необходимость вносить такие изменения. Что может вызывать необходимость их внесения?

Во-первых, язык программирования может быть расширен новыми конструкциями [124]. Например, рассматриваются планы по дальнейшему расширению языка C-light. Большой интерес представляют языковые конструкции, появившиеся в новом стандарте языка C, именуемом C11. Кроме того, C-light может быть расширен языковыми конструкциями родственных языков, таких как Objective C и C++. Во-вторых, представляет интерес создание узкоспециализированных версий ГУК, ориентированных на конкретные классы программ, например, на программы с итерациями, тело которых исполняется один раз для каждого элемента структуры данных конечной размерности. Такая специализация позволяет упростить верификацию. При применении ГУК на практике

важна его расширяемость. Трудно использовать большой ГУК, покрывающий все классы программ. Коллекция специализированных ГУК – лучшее решение данной проблемы [19].

Таким образом, представляют интерес методы, позволяющие упростить разработку нового ГУК. Метод метагенерации УК [151] подходит для решения данных задач.

1.2.1. Формы правил вывода условий корректности

Метод метагенерации УК [151] был предложен Морикони и Шварцем в 1981 году. Метагенератор УК (МГУК) принимает на входе правила вывода аксиоматической семантики в нормальной форме и автоматически порождает ГУК, основанный на вычислении слабейшего предусловия.

Подробное определение нормальной формы приведено в [151]. Морикони и Шварц [151] определяют нормальную форму с помощью пяти ограничений, накладываемых на правило вывода. Кратко опишем результат введения этих ограничений.

Для определения нормальной формы Морикони и Шварц [151] вводят специальную терминологию. В частности, метаидентификаторы, использующиеся в программных фрагментах троек Хоара, они называют *фрагментными переменными*. Используя введенные обозначения, Морикони и Шварц [151] переходят к определению ограничений.

Согласно первому ограничению, все предусловия посылок и постусловие заключения представляют собой предикатные символы, свободно входящие в правило вывода.

Согласно второму ограничению, все свободные предикатные символы каждой формулы из посылки должны входить в объединение множества свободных предикатных символов всех предусловий и постусловий троек Хоара из правила вывода и множества фрагментных переменных из программного фрагмента в заключении.

Согласно третьему ограничению, все фрагментные переменные из про-

граммных фрагментов в посылке должны входить в множество фрагментных переменных из программного фрагмента в заключении.

Четвертое ограничение обеспечивает то, что ГУК сможет вычислить вхождения всех свободных, неинтерпретированных предикатных символов в правилах. В частности, ограничение (4a) устанавливает следующий порядок свободных предикатных символов:

$$\frac{\{P_1\}\ S_1\ \{\mathscr{Q}_1(P_2,...,P_n)\},...,\{P_i\}\ S_i\ \{\mathscr{Q}_i(P_{i+1},...,P_n)\},\quad ...,\ \{P_n\}\ S_n\ \{\mathscr{Q}_n\},\ \Gamma}{\{\mathscr{P}(P_1,...,Q)\}\ S\ \{Q\}}$$

В результате этого устраняются циклические зависимости, такие как посылки вида $\{P\}$ ... $\{P\}$ или пары посылок вида $\{P\}$ ... $\{R\}$ и $\{R\}$ ... $\{P\}$. При таком упорядочивании ограничение (4b) обеспечивает то, что конец каждой цепочки зависимостей (бинарного отношения между предикатными символами A и B, указывающее на то, что связывание A зависит от связывания B) представляет собой либо функцию от постусловия Q, либо связан с программным фрагментом. Это обеспечивается применением к правилам ограничения монотонности, исключающее правила, в которых определенно происходит "изменение знака" между предусловиями в посылке и предусловием в заключении.

Пятое ограничение необходимо для полноты ГУК, т.е. оно гарантирует, что ГУК сможет вычислить слабейшее предусловие $wp(S,Q)$ для данных S и Q.

Необходимо отметить, что ограничения нормальной формы приводят правила вывода к появлению порядка на посылках и к нетипичному виду предусловий в тройках Хоара (если сравнивать их с модифицированным ограничением монотонности). При этом общая форма позволяет избежать нетипичного порядка на посылках. Согласно ограничениям общей формы, предусловия в посылках могут принимать более разнообразный вид и представлять собой не только одиночные предикатные символы, но и формулы предметной области, а также конъюнкцию данных вариантов. Главная идея алгоритма перевода из общей формы в нормальную состоит в следующем: отделяют те предусловия,

которые не являются одиночными предикатными символами. Вместо них будут использоваться "очищенные" предикатные символы. Связь между этими новыми предикатными символами и исходными формулами устанавливается с помощью импликаций, в которых исходные формулы могут быть собраны в конъюнкцию (с удалением дубликатов при необходимости). На конечном шаге алгоритма посылки получившегося правила должны быть переупорядочены для выполнения ограничения (4a).

Отметим, что аксиоматическая семантика языка C-kernel соответствует ограничениям общей формы [124]. По предложенному Морикони и Шварцем [151] алгоритму, правила вывода для C-kernel были переведены в нормальную форму [21]. По получившейся аксиоматической семантике может быть построен полный и корректный (как система вывода) ГУК. Таким образом, метод метагенерации УК можно применить в проекте C-lightVer.

1.2.2. Построение генератора условий корректности по правилам вывода в нормальной форме

Для правила вывода вида

$$\frac{\{P_1\}\ S_1\ \{Q_1\}, ..., \{P_n\}\ S_n\ \{Q_n\},\ \Gamma}{\{P\}\ S\ \{Q\}}$$

слабейшее предусловие (wp) определяется следующим образом:

$$\begin{aligned} wp(S,\ Q) = P[P_1 \leftarrow wp(S_1, Q_1), \ldots, P_n \leftarrow wp(S_n, Q_n)] \wedge \\ (\forall v)\Gamma[P_1 \leftarrow wp(S_1, Q_1), \ldots, P_n \leftarrow wp(S_n, Q_n)] \end{aligned}$$

где $[P_1 \leftarrow wp(S_1, Q_1), ..., P_n \leftarrow wp(S_n, Q_n)]$ обозначают n подстановок, выполненных последовательно слева направо, и v является множеством всех свободных логических переменных в Γ.

1.3. Метод Денни и Фишера анализа условий корректности с помощью семантических меток

При практическом применении дедуктивной верификации могут возникнуть следующие проблемы: программа может быть некорректна, ее спецификации могут быть некорректны, система доказательства теорем может не справиться с доказательством истинных УК, или теория предметной области может быть неполна. В этих случаях пользователь системы верификации получает набор недоказанных условий корректности, но не получает дополнительной информации о причинах неудачи.

Опишем предложенный Денни и Фишером метод семантической разметки [74], ориентированный на такую важную задачу, как анализ, трассировка и объяснение самих УК.

1.3.1. Типы семантических меток

После упрощения УК обычно имеют вид Хорновских дизъюнктов (т.е. $H_1 \wedge H_2 \wedge \ldots \wedge H_n \supset C$). В данном представлении единственное заключение С можно рассматривать как цель. Однако для более осмысленного описания структуры следует дать более детальную характеристику подформул. Ключевая особенность подхода Денни и Фишера [74] состоит том, что различные подформулы располагаются на специальных позициях в правилах Хоара, и, исходя из этого, ГУК добавляет соответствующие метки к УК. Поэтому в модифицированные правила Хоара добавляется семантическая разметка, нужная для объяснения результата применения правила. Метки добавляются к следующим местам: к постусловию посылки, получившемуся при рекурсивном вызове ГУК, к слабейшему предусловию, к сгенерированному УК или к тройке Хоара [114].

Денни и Фишер предложили добавить в правила Хоара [95] семантическую разметку [74] для объяснения результата применения правила. Будем использовать обозначение $\lceil t \rceil^l$, означающее, что терму t сопоставляется метка l. Метки имеют вид $c(o)$, где c – тип метки, o – диапазон строк. Денни и Фишер предложи-

Концепции	Примеры меток	Аспекты условий корректности
Гипотезы • Утверждения • Управляющие предикаты	asm_pre, asm_inv, then, while_t	*Гипотезы* отражают предположения о том, что некоторые логические утверждения выполнены в тех или иных точках программы. Это могут быть как исходные предусловия, так и управляющие выражения операторов наподобие `while` и `if`.
Заключения	ens_post, ens_inv_iter	*Заключения* отражают основное предназначение условий корректности, состоящее в *гарантировании* выполнения тех или иных утверждений в заданных местах программы.
Уточнители • Подстановки • Присваивания	sub, upd, alloc, init	*Уточнители* вводят более детальную характеризацию для гипотез и заключений, отражая способ получения подформулы. Как правило, они соответствуют выражениям и операциям в программе.
Индуктивные уточнители	call, pres_inv	*Индуктивные уточнители* отражают второстепенное предназначение условия корректности. Например, условия корректности для вложенного цикла концептуально связаны с предназначением условий и для охватывающего цикла.

Таблица 1.1. Концепции семантических меток, предложенные Денни и Фишером

ли несколько концепций меток для разных видов подформул из спецификаций и программ [25]. Каждая концепция включает в себя несколько типов меток. Каждому типу метки соответствует текстовый шаблон. Шаблоны используются для генерации объяснений УК.

В таблице 1.1 рассмотрены концепции семантических меток, предложенные Денни и Фишером.

1.3.2. Разметка правил вывода семантическими метками

В качестве примера правила вывода, помеченного семантическими метками, рассмотрим правило вывода для цикла `while`:

$$\frac{\begin{array}{c}\{\lceil P\ \rceil^{asm_pre}\}\ \textbf{prog};\ \{\lceil I\rceil^{ens_inv}\},\\ \lceil\{\lceil I\rceil^{asm_inv}\wedge\lceil B\rceil^{while_t}\}\ \textbf{S}\ \{\lceil I\rceil^{ens_inv_iter}\}\rceil^{pres_inv},\\ \lceil I\rceil^{asm_inv_exit}\wedge\lceil\neg B\rceil^{while_f}\to\lceil Q\rceil^{ens_post}\end{array}}{\{P\}\ \textbf{prog};\ \textbf{while B inv I do S}\ \{Q\}}$$

Отметим, что инвариант, отвечающий за вход в цикл, имеет метку ens_inv. В посылках обособленные подформулы обеих формул, обозначающих выход из цикла и итерацию цикла, имеют соответствующие метки. Также, вся формула в посылке, обозначающая итерацию цикла, имеет метку для обозначения ее второстепенной цели – способствовать сохранению инварианта. В тройке Хоара в посылке постусловие I должно иметь метку, обозначающую ее цель (т.е. гарантирование выполнения инварианта после каждой итерации цикла) для передачи информации при рекурсивном вызове. Более того, все УК, выведенные от данной тройки, должны иметь метку $pres_inv$, обозначающую второстепенную цель такого УК. Это обеспечивается тем, что вся рассматриваемая тройка имеет такую метку. Заметим, как одна и та же формула I имеет четыре различные роли и, поэтому, имеет четыре различные метки [18]. Эта информация следует из контекста, доступного только при применении правила вывода, и не может быть восстановлена простым способом при анализе полученных УК без использования семантической разметки.

1.3.3. Алгоритм генерации текста о соответствии конструкций программ и подформул условий корректности

Метод Денни и Фишера [74] состоит из следующих шагов:

1. Генерация УК с использованием правил вывода с семантической разметкой. При применении такого правила вывода в метках сохраняются но-

мера строк кода исходной программы. В полученных УК подформулы помечены семантическими метками.

2. Задание порядка на семантических метках из УК с помощью создания специального списка меток. Это называется извлечением меток из УК. Метки извлекаются в порядке увеличения номеров соответствующих метке строк.

3. Генерация объяснения УК с помощью последовательного обхода полученного списка меток. Для каждой посещенной при обходе метки текст ее заполненного номерами строк шаблона добавляется к тексту, объясняющему УК.

1.3.4. Пример объяснения условия корректности на естественном языке

В качестве примера применения метода Денни и Фишера рассмотрим верификацию программы поиска максимума в массиве с внесенной ошибкой [19]:

```
1.  // (AND (NEQ a |@NULL|)(> length 0))
2.  int max(int* a, int length)
3.  {
4.      auto int x = 0;
5.      auto int y = length - 1
6.      /* (AND (<= 0 x)(< x length)(<= 0 y)
                (< y length)(>= y x)
                (FORALL (i) (IMPLIES (AND (<= 0 i) (<= i x))
                                     (OR (<= a[i] a[x]) (<= a[i] a[y]))))
                (FORALL (i) (IMPLIES (AND (<= y i)
                                          (<= i (- length 1)))
                                     (OR (<= a[i] a[x])
                                     (<= a[i] a[y]))))) */
7.      while (x == y)
8.      {
9.          if (a[x] <= a[y]) {x = x + 1; } else {y = y - 1;}
10.     }
```

```
11.     return x;
12. }
13. /* (AND (<= 0 x)(< x length)
            (FORALL (i) (IMPLIES (AND (<= 0 i) (< i length))
                                 (>= a[x] a[i])))) */
```

Спецификации данной программы записаны в префиксной форме, где конструкции `AND`, `OR`, `IMPLIES` и `FORALL` языка спецификаций представляют собой логическое и, или, импликацию и квантор всеобщности соответственно. Спецификации, расположенные перед циклом, а также до и после определения функции, являются инвариантом цикла, а также предусловием и постусловием функции соответственно.

Ошибка в программе заключается в использовании оператора `==` в условии цикла `while` вместо оператора `!=`. Поэтому, было получено недоказанное условие корректности:

$$
\left(
\begin{array}{l}
\left[
\begin{array}{l}
0 \leq MD(MeM(x)) \wedge MD(MeM(x)) < MD(MeM(length)) \wedge \\
0 \leq MD(MeM(y)) \wedge \\
MD(MeM(y)) < MD(MeM(length)) \wedge \\
MD(MeM(y)) \geq MD(MeM(x)) \wedge \\
\forall i \left(
\begin{array}{l}
0 \leq MD(MeM(i)) \wedge MD(MeM(i)) \leq MD(MeM(x)) \\
\Rightarrow \left(
\begin{array}{l}
MD(MeM(a), MD(MeM(i))) \leq MD(MeM(a), MD(MeM(x))) \vee \\
MD(MeM(a), MD(MeM(i))) \leq MD(MeM(a), MD(MeM(y)))
\end{array}
\right)
\end{array}
\right) \wedge \\
\forall i \left(
\begin{array}{l}
MD(MeM(y)) \leq MD(MeM(i)) \wedge \\
MD(MeM(i)) \leq MD(MeM(length)) - 1 \\
\Rightarrow \left(
\begin{array}{l}
MD(MeM(a), MD(MeM(i))) \leq MD(MeM(a), MD(MeM(x))) \vee \\
MD(MeM(a), MD(MeM(i))) \leq MD(MeM(a), MD(MeM(y)))
\end{array}
\right)
\end{array}
\right)
\end{array}
\right]^{ass_inv_exit(6)} \wedge \\
\left[cast \left(
\begin{array}{l}
val(val(MD(MeM(x)) = MD(MeM(y))), MeM...STD)), \\
type(MD(MeM(x)) = MD(MeM(y))), MeM, TP), \mathsf{int}
\end{array}
\right) = 0 \right]^{while_ff(7)}
\end{array}
\right)
$$

$$
\Rightarrow \left[
\begin{array}{l}
0 \leq MD(MeM(x)) \wedge MD(MeM(x)) < MD(MeM(length)) \wedge \\
\forall i \left(
\begin{array}{l}
0 \leq MD(MeM(i)) \wedge MD(MeM(i)) < MD(MeM(length)) \\
\Rightarrow MD(MeM(a), MD(MeM(x))) \geq MD(MeM(a), MD(MeM(i)))
\end{array}
\right)
\end{array}
\right]^{ens_post(13)}
$$

где отображения MeM и MD относятся к модели памяти С-программы, описанной в разделе 1.5. Отметим, что данное УК снабжено семантическими метками. Продемонстрированное УК является слишком сложным для анализа "вручную". В результате генерации объяснения по данному УК получается следующий текст:

```
This VC corresponds to function "max".
```

```
Hence, given
- assumption that loop invariant holds without loop entry at line 6,
- assumption that the loop condition doesn't hold at line 7
        ensure that postcondition goal from line 13 holds
```

Рассматриваемое УК означает, что из выполнения инварианта и невыполнения условия цикла следует постусловие. Таким образом, данная формула соответствует выходу из цикла. Соответственно, рассматриваемый текст явно указывает на то, что необходимо обратить внимание на допущения о выполнении инварианта цикла (строка 6) и невыполнении условия цикла (строка 7). При детальном рассмотрении допущения о невыполнении условия цикла можно прийти к пониманию того, что есть проблема с осуществлением входа в цикл (строка 7). Следовательно, для обнаружения ошибки необходимо рассмотреть условие цикла. Заметим, что именно неправильное условие цикла представляет собой искомую ошибку. Таким образом данный текст содержит информацию, которая помогает локализовать ошибку.

1.4. Символический метод верификации финитных итераций

Рассмотрим символический метод верификации финитных итераций, позволяющий избежать задания инвариантов для циклов специального вида [158].

1.4.1. Финитные итерации

Рассмотрим итерацию над последовательностью данных, тело которой состоит из последовательности операторов присваивания и условных операторов. Представим его в виде оператора векторного (т.е. одновременного) присваивания $v = body(v, x)$, где x – параметр итерации, v – вектор остальных переменных, $body(v, x)$ – вектор условных выражений, построенных с помощью операции if-$then$-$else$. Такое представление получается посредством последовательности подходящих подстановок, которые осуществляют замену условных

операторов условными выражениями и последовательности операторов присваивания одним оператором присваивания.

Пусть $memb(S)$ обозначает мультимножество элементов структуры S и $empty(S) = true$ тогда и только тогда, когда $|memb(S)| = 0$. Определим

1. $choo(S)$ возвращает некоторый элемент из $memb(S)$, если $\neg empty(S)$;

2. $rest(S) = S'$, где $memb(S') = memb(S) \setminus \{choo(S)\}$, если $\neg empty(S)$.

Финитная итерация соответствует виду:

$$\mathbf{for\ x\ in\ S\ do\ v := body(v, x)} \tag{1.1}$$

где S является структурой данных, x является переменной типа “элемент S”, v является вектором переменных цикла без x, $body$ представляет тело цикла, которое не изменяет x и завершается для каждого $x \in S$.

1.4.2. Представление финитных итераций с помощью операций замены

Пусть v_0 обозначает начальные значения переменных из v. Определим операцию замены $rep(v, S, body)$ для цикла:

1. если $empty(S)$, то $rep(v_0, S, body) = v_0$;

2. если $\neg empty(S)$, то $rep(v_0, S, body) = body(rep(v_0, rest(S), body), choo(S))$.

Пусть $R(y \leftarrow exp)$ обозначает результат подстановки выражения exp вместо всех вхождений переменной y в формулу R. Пусть $R(vec \leftarrow vexp)$ обозначает результат одновременной подстановки в формулу R компонент вектора выражений $vexp$ вместо всех вхождений соответствующих компонент вектора vec. Следующая теорема [158], сформулированная и доказанная Валерием Александровичем Непомнящим, описывает полезные свойства операции замены:

Теорема. Итерация 1.1 эквивалентна операции векторного присваивания $v := rep(v_0, S, body)$.

Следствием из данной теоремы является следующее правило вывода УК:

$$\frac{\{P\}\ prog\ \{Q(v \leftarrow rep(v_0, s, body))\}}{\{P\}\ prog;\ \ \texttt{for}\ \ x\ \ \texttt{in}\ \ S\ \ \texttt{do}\ \ v\ \ =\ \ body(v,\ \ x)\ \ \texttt{end}\ \{Q\}}$$

где P – предусловие, Q – постусловие, независящее от параметра итерации x, $prog$ – фрагмент программы, а $\{P\}$ $prog$ $\{Q\}$ обозначает частичную корректность программы $prog$ относительно P и Q.

В результате применения правила порождаются УК, содержащие операцию замены. Для доказательства таких УК применяется как универсальная техника, базирующаяся на принципах индукции, так и проблемно-ориентированная техника [118].

1.4.3. Генерация операции замены для финитных итераций без вложенных условных инструкций над неизменяемыми массивами

Символический метод верификации финитных итерации включает четыре случая:

1. Финитная итерация над неизменяемыми структурами данных без выхода из цикла.

2. Финитная итерация над неизменяемыми структурами данных с выходом из цикла.

3. Финитная итерация над изменяемыми структурами данных без / с выходом из цикла.

4. Финитная итерация над иерархическими структурами данных с выходом из цикла.

Алгоритм генерации операции замены [146] был предложен для финитной итерации с инструкцией `break` над неизменяемым массивом без вложенных условных инструкций.

Введем специальную процедуру. Данная процедура генерирует векторное равенство значения функции rep и значений компонент вектора v. Процедура основана на символьном вычислении значений компонент вектора v и принимает в качестве аргумента последовательность присваиваний вида

$$\{\mathbf{x_1 = expr_1(x_1, x_2, \ldots, x_k); x_2 = expr_2(x_1, x_2, \ldots, x_k); \ldots x_k = expr_k(x_1, x_2, \ldots, x_k);}\}$$

где $expr_j(j = 1, 2, \ldots k)$ — выражения языка C-kernel. Процедурой выполняются подстановки

$$x_1 = expr_1(x_1, x_2, \ldots, x_k);$$
$$x_2 = expr_2(expr_1(x_1, x_2, \ldots, x_k), x_2, \ldots, x_k);$$
$$\ldots$$
$$x_k = expr_k(expr_1(x_1, x_2, \ldots, x_k), expr_2(expr_1(x_1, x_2, \ldots, x_k), x_2, \ldots, x_k), \ldots, x_k);$$

Так как значение переменных правой части является результатом исполнения предыдущей итерации цикла, то для каждой переменной x_m $(1 \leq m \leq k)$ правой части присваиваний выполним подстановку m-й компоненты значения $rep(i-1, v, S, body)$. Полученные таким образом правые части присваиваний переменным $(x_1,\ x_2, \ldots,\ x_k)$ обозначим как $(z_1,\ z_2, \ldots,\ z_k)$ соответственно. Так как $rep(i, v, S, body) = (x_1,\ x_2, \ldots,\ x_k)$, то генерируется равенство вида

$$rep(i, v, S, body) = (z_1,\ z_2, \ldots,\ z_k)$$

Рассмотрим далее алгоритм генерации операции замены. Данный алгоритм основан на генерации аксиом, определяющих функцию rep. Пусть вектор v представляет собой вектор переменных $(x_1,\ x_2, \ldots,\ x_k)$.

Если $v_0 = (y_1,\ y_2, \ldots,\ y_k)$, то генерируется аксиома

$$rep(0, v, S, body) = (y_1,\ y_2, \ldots,\ y_k)$$

Если тело цикла представляет собой последовательность присваиваний, то применяется специальная процедура. Алгоритм генерирует аксиому, которая является порожденной специальной процедурой формулой под кванторами по переменным вектора v. Рассмотрим случаи, когда тело цикла содержит условные инструкции, которые не вложены друг в друга.

Для каждой инструкции if вида $if\,(e(i, x_1, x_2, \ldots, x_k))\{A;\}\, else\,\{B;\}$, где A и B являются составными выражениями, содержащими операции присваивания, генерируются две аксиомы:

$$\forall x_1 \forall x_2 \ldots \forall x_k\ e(i, x_1, x_2, \ldots, x_k) \Rightarrow A^* \quad \forall x_1 \forall x_2 \ldots \forall x_k\ \neg e(i, x_1, x_2, \ldots, x_k) \Rightarrow B^*$$

где A^* and B^* являются результатами применения специальной процедуры к A и B соответственно.

Инструкция `break` может находиться на верхнем уровне тела цикла. Этот случай означает, что цикл исполняется не больше одного раза и все инструкции после `break` в теле цикла игнорируются. В таком случае функция rep определяется для $i = 0, 1$.

Второй случай означает, что для некоторого j, такого что $0 < j \leq n$, происходит выход из цикла и такое j определено условием инструкции if. Обозначим как A ту ветвь инструкции if, в которой находится `break`. В этом случае генерируется аксиома $\forall x_1 \forall x_2 \ldots \forall x_k\ e(i, x_1, x_2, \ldots, x_k) \Rightarrow (A^* \wedge (\forall l\, i < l \Rightarrow A^*))$, где A^* является результатом применения к A специальной процедуры. В случае, когда инструкция `break` расположена в блоке `else`, используется отрицание e.

1.5. Языки C-light и C-kernel

Наличие формальной семантики для языка программирования является необходимым условием разработки метода верификации программ, написанных на нем. Для языка C невозможно разработать компактную формальную семантику, поэтому в проекте по автоматической верификации C-программ лаборатории теоретического программирования ИСИ СО РАН используется представительное подмножество языка, которое называется C-light [156]. Для большинства императивных языков программирования, в т.ч. и для C-light, наиболее естественным подходом при конструировании семантики является операционный. Однако процесс верификации, основанный на подобной семантике, весьма

сложен, и для верификации наиболее пригодной является аксиоматическая семантика.

Аксиоматическая семантика языка C-light была бы слишком громоздка для практического применения, поэтому был предложен новый двухуровневый метод верификации программ. На первом этапе C-light транслируется в промежуточный язык C-kernel [157] с целью элиминации некоторых конструкций языка C-light, трудных для аксиоматической семантики. Для трансляции используется набор формальных правил [147]. Отметим, что в качестве языка спецификаций был выбран язык ACSL [46], обретающий популярность в данной области в последнее время. Спецификации на данном языке оформляются как комментарии на языке C, и, поэтому, не могут влиять на семантику самой программы. В ACSL-секциях `requires` записываются конъюнкты предусловия, в ACSL-секциях `ensures` записываются конъюнкты постусловия. На втором этапе для промежуточной C-kernel программы генерируются условия корректности, которые в дальнейшем передаются на блок доказательства. Таким образом, аннотированная C-kernel программа является промежуточным представлением аннотированной исходной программы в рассматриваемой системе верификации.

1.5.1. Трансляция из C-light в C-kernel

Входным языком системы C-lightVer является язык C-light [156]. Этот язык является представительным подмножеством языка C99 [164]. Отметим, что в проекте был принят подход, основанный на постепенном расширении входного языка. На текущий момент не моделируется раскладка битов в памяти, поэтому в языке C-light не поддерживаются побитовые операции. Также не поддерживаются указатели на функции. Для языка C-light была разработана операционная семантика. Модель памяти, основанная на отображениях MeM и MD, используется в этой семантике. MeM является отображением из имен объектов в их адреса, MD является отображением из адресов объектов в их значения.

Так как в операционной семантике языка C-light не моделируется низко-

уровневые операции с памятью, то в качестве адресов используются объекты соответствующего класса, на которых определены только три операции: равенство, неравенство и создание нового адреса, не равного ни одному другому адресу. Проверка неравенства нового адреса осуществляется с помощью функции Dom, возращающей область определения отображения.

Операция *upd* позволяет создавать новое отображение MD, когда изменяется состояние памяти. Определим значение выражения $upd(MD,\ addr,\ val)$, где MD является отображением $address \to value$, $addr$ является адресом и val является значением. Если MD содержит пару $(adr\ val')$ (где val' является некоторым значением), то отображение $upd(MD,\ addr,\ val)$ отличается от MD заменой пары $(adr\ val')$ на пару $(adr\ val)$. Если $addr$ не принадлежит области определения MD, тогда отображение $upd(MD,\ addr,\ val)$ отличается от MD добавлением пары $(adr\ val)$.

Отображения MeM и MD определены набором следующих аксиом:

1. $MeM(obj) \neq NULL$
2. $MeM(obj_1) \neq MeM(obj_2), obj_1 \neq obj_2$
3. $MeM(a, 0) = MeM(a)$
4. $MeM(a, i) \neq MeM(obj), a[i] \neq obj$
5. $upd(MD,\ NULL,\ val) = MD$
6. $upd(MeM,\ obj,\ NULL) = MeM$
7. $remove(MD, NULL) = MD$
8. $(upd(MD,\ addr,\ val))(addr) = val,\ addr \neq NULL$
9. $(upd(MD, addr_1, val))(addr_2) = MD(addr_2),\ addr_1 \neq addr_2$
10. $upd(MD,\ MeM(obj),\ MD(MeM(obj))) = MD$
11. $upd(MeM,\ obj\ MeM(obj)) = MeM$
12. $(upd(MeM, obj, addr))(obj) = addr$
13. $(upd(MeM, obj_1, addr))(obj_2) = MeM(obj_2),\ obj_1 \neq obj_2$
14. $(remove(MD, addr_1))(addr_2) = MD(addr_2),\ addr_1 \neq addr_2$

15. $(remove(MeM, obj_1))(obj_2) = MeM(obj_2),\ obj_1 \neq obj_2$

16. $remove(upd(MD, addr, val), addr) = MD$

17. $remove(upd(MeM, obj, addr), obj) = MeM,\ addr \neq NULL$

Язык C-kernel является очень ограниченным подмножеством C-light [157]. Для языка C-kernel определена аксиоматическая семантика. Это позволяет верифицировать C-kernel программы. Трансляция из C-light в C-kernel является первой стадией исполнения системы C-lightVer. Эта трансляция основана на наборе правил. Эти правила определяют трансляцию различных конструкций C-light в эквивалентные конструкции C-kernel.

Главной целью этой трансляции является локализация побочных эффектов. Поэтому, правила трансляции выносят сложные подвыражения (не переменные и константы) из выражений и операторов, используя задание вспомогательных переменных со значениями этих подвыражений. В итоге, все инструкции и выражения транслируются в форму, где только переменные и константы являются их аргументами. В качестве примера рассмотрим правило трансляции Ops: если e_i не является переменной или константой, $e_{i+1},\ \ldots,\ e_n$ – переменные или константы, f – функция или $+$, $-$, $*$, $/$, $<$, $>$, $<=$, $=>$, $!=$, $==$, тогда $f(e_1,\ \ldots,\ e_{i-1},\ e_i,\ e_{i+1},\ \ldots,\ e_n)$ транслируется в

$$(x\ =\ e_i,\ f(e_1,\ \ldots,\ e_{i-1},\ x,\ e_{i+1},\ \ldots,\ e_n))$$

где x является новой переменной с тем же типом, что и e_i.

Из-за ограничений языка C-kernel левая часть в инструкциях присваивания может принимать только одну из следующих форм:

1. var

2. $*pointer_var$

3. $var.structure_field$

4. $(*pointer_var).structure_field$

5. $array[index]$

6. $*(array_of_pointers[index])$

Это позволяет упростить вид объектов-аргументов MeM.

Так как язык C-kernel является подмножеством языка C-light, то его операционная семантика является той же самой, что и семантика языка C-light. Это позволило доказать, что правила трансляции сохраняют эквивалентность [31]. УК полученной C-kernel программы генерируются на втором этапе исполнения системы C-lightVer.

1.5.2. Смешанная аксиоматическая семантика языка C-kernel

Аксиоматическая семантика языка C-kernel основана на методе слабейшего предусловия.

Для применимости метода метагенерации в первую очередь необходимо убедиться, что аксиоматическая семантика языка C-kernel удовлетворяет ограничениям общей формы [151]. Для этого был проведен соответствующий анализ семантики [21]. Полный обзор аксиоматической семантики языка C-kernel можно найти в [31]. Анализ показал, что ограничениям нормальной формы данная семантика не удовлетворяет, а ограничениям общей формы — удовлетворяет. Соответственно, с помощью предложенного Морикони и Шварцем [151] алгоритма был осуществлен перевод данной семантики в нормальную форму.

Анализировалась не только непосредственно аксиоматическая семантика языка C-kernel, но и семантика программ символического метода [158] и программ инженерной математики [32]. Анализ показывает, что при расширении языка такими конструкциями, как финитные итерации над структурами данных, особых препятствий не возникает, т.е. они тоже могут вкладываться в ограничения общей формы.

Рассмотрим правило вывода в нормальной форме для пустой программы:

$$\frac{P \to Q}{\{P\}\ \mathbf{emptyProgram}\ \{Q\}}$$

где $emptyProgram$ – обозначение пустой программы. Слабейшее предусловие для такой программы задано следующим образом: $wp(emptyProgram,\ Q) = Q$

Рассмотрим правило вывода в нормальной форме для декларации пере-

менной:

$$\frac{\{P\}\ \mathbf{prog};\ \{Q(MeM \leftarrow upd(MeM, var, addr))\},\ addr \notin Dom(MD)}{\{P\}\ \mathbf{prog};\ \mathbf{type\ var};\ \{Q\}}$$

где $addr$ – новый объект класса адрес, не равный ни одному предыдущему адресу.

Рассмотрим правило вывода в нормальной форме для присваивания переменной:

$$\frac{\{P\}\ \mathbf{prog};\ \{Q(MD \leftarrow upd(MD, MeM(var), rval))\}}{\{P\}\ \mathbf{prog};\ \mathbf{var} = \mathbf{rval}\ \{Q\}}$$

Слабейшее предусловие для присваивания элементу массива задано следующим образом: $wp(var = rval,\ Q) = Q(MD \leftarrow upd(MD, MeM(var), rval))$

Рассмотрим правило вывода в нормальной форме для присваивания элементу массива:

$$\frac{\{P\}\ \mathbf{prog};\ \{Q(MD \leftarrow upd(MD, MeM(a, i), rval))\}}{\{P\}\ \mathbf{prog};\ \mathbf{a[i]} = \mathbf{rval}\ \{Q\}}$$

Слабейшее предусловие для присваивания элементу массива задано следующим образом: $wp(a[i] = rval,\ Q) = Q(MD \leftarrow upd(MD, MeM(a, i), rval))$

Метод смешанной аксиоматической семантики [147] позволяет использовать специальные версии правил вывода для определенных программных конструкций. Отметим, что в C-программах есть переменные, к которым не применяются операции взятия адреса и разыменования указателя. Количество таких переменных в C-программах может быть значительным. Для таких переменных может быть использована более простая модель памяти, чем модель, основанная на MeM и MD. Следовательно, более простые правила вывода могут быть применены к инструкциям над такими переменными. Это позволяет упростить УК. Рассмотрим правило вывода для присваивания такой переменной:

$$\frac{\{P\}\ \mathbf{prog};\ \{Q(var \leftarrow rval))\}}{\{P\}\ \mathbf{prog};\ \mathbf{var} = \mathbf{rval}\ \{Q\}}$$

Слабейшее предусловие в таком случае задано следующим образом:

$$wp(var = rval,\ Q) = Q(MD \leftarrow upd(MD, MeM(var), rval))$$

Рассмотрим упрощенное с помощью смешанной аксиоматической семантики правило для присваивания элементу массива:

$$\frac{\{P\}\ \mathbf{prog};\ \{Q(a \leftarrow upd(a, i, rval))\}}{\{P\}\ \mathbf{prog};\ \mathbf{a[i] = rval}\ \{Q\}}$$

Слабейшее предусловие для присваивания элементу массива задано следующим образом:

$$wp(a[i] = rval,\ Q) = Q(a \leftarrow upd(a, i, rval))$$

Рассмотрим правило вывода в общей форме для условной инструкции if:

$$\frac{\{P \wedge B\}\ \mathbf{prog};\ \mathbf{S_1}\ \{Q\},\ \{P \wedge \neg B\}\ \mathbf{prog};\ \mathbf{S_2}\ \{Q\}}{\{P\}\ \mathbf{prog};\ \mathbf{if\ B\ S_1\ else\ S_2}\ \{Q\}}$$

Рассмотрим правило вывода для локальных инвариантов (утверждений):

$$\frac{\{P\}\ \mathbf{prog}\ \{INV \rightarrow Q\}}{\{P\}\ \mathbf{prog};\ \{INV\}\ \mathbf{emptyProgram}\ \{Q\}}$$

где INV — локальный инвариант (утверждение).

Рассмотрим правило вывода для условной инструкции if, заданное с помощью локальных инвариантов (утверждений):

$$\frac{\{P\}\ \mathbf{prog}\ \{B\}\ \mathbf{S_1}\ \{Q\}, \{P\}\ \mathbf{prog}\ \{\neg B\}\ \mathbf{S_2}\ \{Q\}}{\{P\}\ \mathbf{prog};\ \mathbf{if\ B\ S_1\ else\ S_2}\ \{Q\}}$$

Запишем это правило вывода в нормальной форме:

$$\frac{\{P\}\ \mathbf{prog}\ \{B \rightarrow wp(S_1,\ Q)) \wedge (\neg B \rightarrow wp(S_2,\ Q))\}}{\{P\}\ \mathbf{prog};\ \mathbf{if\ B\ S_1\ else\ S_2}\ \{Q\}}$$

Слабейшее предусловие для инструкции if задано следующим образом:

$$wp(if\ B\ S_1\ else\ S_2,\ Q) = (B \rightarrow wp(S_1,\ Q)) \wedge (\neg B \rightarrow wp(S_2,\ Q))$$

Рассмотрим правило вывода в общей форме для цикла *while*:

$$\frac{\{P\}\ \mathbf{prog};\ \{I\},\ \{I \wedge B\}\ \mathbf{S}\ \{I\},\ I \wedge \neg B \to Q}{\{P\}\ \mathbf{prog};\ \mathbf{while\ B\ inv\ I\ do\ S}\ \{Q\}}$$

Запишем правило вывода в нормальной форме для цикла *while*:

$$\frac{\{P\}prog\{I\ \wedge\ (\forall y(B \wedge I \to wp(S,I))[v \leftarrow y])\ \wedge\ (\forall y(\neg B \wedge I \to Q)[v \leftarrow y])\}}{\{P\}\ \mathbf{prog};\ \mathbf{while\ B\ inv\ I\ do\ S}\ \{Q\}}$$

где v – переменные цикла, y – новые переменные, не используемые в верифицируемой программе.

Слабейшее предусловие для цикла *while* задано следующим образом:

$$wp(while\ B\ inv\ I\ do\ S,\ Q) = I \wedge$$
$$(\forall y(B \wedge I \to wp(S,I))[v \leftarrow y]) \wedge (\forall y(\neg B \wedge I \to Q)[v \leftarrow y])$$

1.6. Модуль трансляции из C-light в C-kernel

Для реализации модуля трансляции из C-light в C-kernel была использована связка LLVM/Clang [147]. Во-первых, опишем такую часть транслятора, как реализацию проверки на применимость смешанной аксиоматической семантики [8]. Реализацию такой проверки позволило упростить использование API Clang. Сначала составляется список всех переменных данной программы. Потом каждая переменная из этого списка проверяется на применение к ней операции взятия адреса. Такая проверка реализована путем обхода дерева кода (AST) с использованием наследником класса, метод `visitStmt` которого применяется к каждому узлу дерева. У этого наследника заведено булевское поле `check`, присвоено ему значение "истина" и переопределен виртуальный метод `visitStmt(Stmt* S)` таким образом, чтобы его реализация, соответствовала псевдокоду:

```
Если S in UnarOp, то{
    Если S == '&', то{
        Выражению Res = аргумент S;
```

```
Если Res является обращением к переменной, то{
    Переменная Var = (Переменная)Res;
        Если Var == рассматриваемая переменная, то{
            check = false;
            Остановить обход AST;}}}}
```

В вышеприведенном псевдокоде `UnarOp` — это множество унарных операций. Если после окончания обхода AST поле `check` будет иметь значение "истина", то проверка показала, что к рассматриваемой переменной не применяются операция взятия адреса. Таким образом транслятор создает список искомых объектов, для доступа к значениям которых можно не применять композицию MeM и MD. Отметим, что это не весь список объектов, к которым теоретически можно применить метод смешанной аксиоматической семантики. То есть, данная проверка корректна, но не полна.

Во-вторых, опишем такую часть транслятора, как реализацию процедуры, позволяющей от конструкций промежуточной C-kernel программы вернуться к конструкциям исходной C-light программы. Для реализации такой процедуры транслятор добавляет в код аннотированной C-kernel программы дополнительную информацию [147], которая позволяет от этого кода вернуться к исходной программе. Эта дополнительная информация добавляется после каждого применения правила трансляции и имеет следующую структуру:

- перед участком кода, в котором были произведены изменения, пишется C-комментарий, содержащий следующую информацию:

    1. `begin changes` — информация, указывающая на то, что перед этим участком кода было применено правило трансляции;
    2. `id` — уникальный идентификатор правила трансляции;
    3. `count` — целое число, содержащее номер применения правила трансляции. Перед началом применения правил трансляции этот число равно `1`. После каждого применения любого из правил трансляции этот номер увеличивается на `1`;

4. `interval` — диапазон номеров строк участка программы, начинающегося с комментария `/* begin changes */` и кончающегося соответствующим комментарием `/* end changes */`. После каждого применения любого из правил трансляции все поля `interval` во всей дополнительной информации обновляются. В дополнительной информации используется только один диапазон, так как правила трансляции предписывают переписывать только локальные участки кода.

- после участка кода, в котором были произведены изменения, пишется C-комментарий, содержащий слова `end changes`.

Рассмотрим пример добавления дополнительной информации при применении одного из правил трансляции. Код до применения правила трансляции:

```
56. if (k++)
57. {
58.     j++;
59. }
```

Здесь числа `56-59` — номера строк, добавленные для наглядности. Код после применения правила трансляции:

```
56. /* begin changes Norm6 17 56-60 */ int i = k++;
57. if (i)
58. {
59.    j++;
60. } /* end changes */
```

Числа `56-60` — номера строк, добавленные для наглядности. Здесь

- `/* begin changes Norm6 17 56-60 */` и `/* end changes */` – комментарии, содержащие дополнительную информацию;
- `Norm6` — это `id`, уникальный идентификатор данного правила трансляции;
- `17` — это `count`, номер применения правила трансляции;

- `56-60` — это `interval`, то есть диапазон номеров строк.

  Применение процедуры, позволяющей от конструкций промежуточной C-kernel программы вернуться к конструкциям исходной C-light программы, проиллюстрировано примером, описанным в разделе 4.3.2.

1.7. Модуль метагенерации условий корректности

Правила в нормальной форме, поступающие на вход метагенератору, задают собой шаблоны для сопоставления с программными конструкциями [124]. Важной задачей является разработка для них некоторого языка. Отметим, что в основе этого языка лежит язык логики первого порядка и целевой язык программирования. Вместе с тем, в этот язык необходимо добавить некоторые метаобозначения, поскольку правила вывода задают семантику не конкретных программ, а схем программ. Заметим, что в соответствии с идеей метагенерации, в общем случае разрабатывается схема языка шаблонов. Рассмотрим случай применения этой схемы для языка C-light.

1.7.1. Язык для задания шаблонов

Как было сказано выше, основой языка шаблонов для C-light является логика первого порядка и грамматика языка C. Введение метаобозначений в этот язык фактически означает, что в выражениях этого языка сохраняются некоторые нетерминальные символы (например, неинтерпретированные предикатные символы и фрагментные переменные [151] для обозначения фрагментов кода). Принадлежность метаданных тому или иному классу в языке шаблонов задается в явном виде [129]. В языке шаблонов не устанавливается жестких ограничений на то, что символы P, Q и R должны обозначать предикаты, а S_i – программные фрагменты. Для обозначения данных конструкций в языке шаблонов могут быть использованы любые символы, а принадлежность к определенному классу конструкций в случае необходимости указывается явно. Например, конструкция `any_code(S)` может соответствовать любой последовательности

(включая пустую) конструкций языка программирования, а `exists_code(S)` соответствует одиночной конструкции. Конструкция `simple_expression` обозначает выражение, которое не содержит побочных эффектов.

Пусть `construction` – обозначение одной из специальных синтаксических конструкций и пусть `construction_identifier` обозначает уникальный идентификатор. Запись `construction(construction_identifier)` в шаблоне означает, что у данного вхождения `construction` вводится идентификатор `construction_identifier`. Программная конструкция, сопоставленная такому вхождению `construction`, будет обозначаться как `construction_identifier`. Введение такого обозначения необходимо для того, чтобы описать некоторые правила вывода. Например такие, у которых программная конструкция, сопоставленная такому вхождению `construction`, используется в их посылках.

Проиллюстрируем это правилом вывода для цикла `while`

```
{P} prog {INV},
    {INV /\ B} S {INV},
INV /\ (not B) => Q
|-
{any_predicate(P)} any_code(prog)
{any_predicate(INV)}
while(simple_expression(B)) any_code(S)
{any_predicate(Q)}
```

где /\ обозначает конъюнкцию, а => обозначает импликацию.

1.7.2. Реализация метагенератора

При рассмотрении реализации метагенератора необходимо обратить внимание на разницу между ней и идеями Мориконы и Шварца [151]. Метагенератор в системе C-lightVer представляет собой функцию с двумя параметрами [19], то есть, при его работе совершается каррирование. Таким образом, если H – аксиоматическая семантика и AP – аннотированная программа, которую нужно верифицировать, то $\mathrm{MetaVCG}(H, AP) = \mathrm{VCG}_H(AP)$, где MetaVCG – МГУК,

а VCG_H – ГУК, построенный по H. Это позволяет проводить эксперименты по верификации самого метагенератора и верифицировать одну программу вместо двух, одна из которых порождалась бы без спецификаций.

1.7.3. Сопоставление программных конструкций и шаблонов

Аргументы метагенератора – аксиомы и правила вывода вместе с аннотированной программой – анализируются и транслируются в соответствующее внутреннее представление. Отметим, что для этого используется в том числе API, предоставляемое компилятором Clang [18].

Так как API Clang имеет встроенный лексический и синтаксический анализатор C-программ, то использование API Clang позволяет облегчить реализацию этой функциональности. На первом этапе, анализатор с помощью инструментария, предоставляемого API Clang, строит внутреннее представление шаблона. Данное представление ориентировано на язык C++. Так как важной задачей проекта C-light является частичная самоприменимость, то внутреннее представление на следующем шаге переводится в структуру `pattern_node`, совместимую с C-light. Внутренним представлением спецификаций является дерево, образованное структурой `term_node`. В результате работы анализатора шаблонов получается внутреннее представление шаблонов – множество деревьев, узлами которых является структуры `pattern_node`.

Тип данных `pattern_node` представляет аксиомы и заключения правил вывода. Такая структура задает дерево, первый и последний узел которого является соответственно предусловием и постусловием. Каждый узел имеет атрибуты (категория, идентификатор, тип), которые активно используются в процессе сопоставления данному шаблону программных конструкций. Также одним из атрибутов является таблица соответствия между идентификаторами программы и шаблона, которая заполняется в процессе сопоставления. Внутренним представлением программы служит древовидная структура `program_node`, подобная структуре `pattern_node`.

Таким образом, метагенератор строит деревья как для аннотированной

программы, так и для набора аксиом и правил вывода. В соответствии с направлением вывода для правейшей/левейшей программной конструкции происходит вывод соответствующего ей заключения правила вывода. Для выбранного шаблона процедура рекурсивно применяется к его посылкам.

Необходимо отметить, что задача сопоставления программной конструкции и шаблона правила вывода решается с помощью алгоритма сопоставления деревьев [21]. На текущий момент он реализован как "жадный" алгоритм. Корректность его применения гарантируется простотой аксиоматической семантики языка C-kernel [157] и ограничениями языка C-light (например, запретом на передачу управления в составные операторы извне) [31]. Аксиоматическая семантика правил вывода для финитных итераций над последовательностями данных [117] также позволяет применять данный алгоритм.

1.8. Модуль доказательства условий корректности

В качестве модуля доказательства УК в системе C-lightVer используются различные системы доказательства теорем: Simplify [75], Z3 [51, 73], CVC4 [43], PVS [111, 160], ACL2 [98, 110, 150]. Рассмотрим критерии выбора системы для доказательства условий корректности, в результате анализа которых была выбрана система ACL2.

1.8.1. Критерии выбора системы доказательства

Во-первых, в проекте C-lightVer максимальное предпочтение отдается методам и системам, для которых доказана корректность [31]. Поэтому, при выборе системы для доказательства УК в проекте C-lightVer отдается предпочтение формально верифицированным системам.

Во-вторых, в результате применения символического метода верификации финитных итераций в УК появляются применения рекурсивных функций, соответствующих циклам [158]. Это приводит к необходимости доказательства, основанного на индукционных схемах, соответствующих определениям таких рекурсивных функций. Поэтому, при выборе системы для доказательства УК

в проекте C-lightVer отдается предпочтение системам, где реализована автоматизация доказательства по индукции.

Рассмотрим теперь типы систем доказательства теорем и их соответствие описанным критериям. Отметим, что можно выделить два основных класса систем доказательства теорем [155]: автоматизированные и интерактивные. Главным отличием между этими типами систем является стиль доказательства. В автоматизированных системах обычно выполняется автоматический поиск подходящего доказательства. В интерактивных системах доказательство предоставляется пользователем, который сообщает, какие тактики доказательства применить.

Важным классом автоматизированных систем доказательства являются SMT-решатели. Такие системы решают задачи выполнимости для формул в различных теориях, используя решение задачи выполнимости булевых формул в конъюнктивной нормальной форме (КНФ). Отметим, что такая задача является NP-полной. Трудность решения такой задачи с помощью перебора всех вариантов приводит к необходимости использовать эвристики. Такой эвристикой является перебор с возвратом, являющийся основой алгоритма Дэвиса—Патнема–Логемана–Лавленда (DPLL). Этот алгоритм реализован во многих SMT-решателях, например Z3 [51, 73], CVC4 [43]. Для поддержки кванторов в SMT-решателях реализуют специальные алгоритмы сопоставления. Одним из первых SMT-решателей, где был реализован алгоритм сопоставления для поддержки кванторов, является Simplify [75].

В C-lightVer применялись SMT-решатели, а именно: Simplify, Z3 и CVC4. Кроме автоматизации, другим важным преимуществом SMT-решателей является генерация контрпримера в случае невыполнимости формулы. Это позволяет находить контрпримеры к условиям корректности в случае наличия ошибок в программе.

Так как при доказательстве УК проверяется не выполнимость, а истинность УК, то на вход SMT-решателей подается отрицание УК и решается задача невыполнимости отрицания УК. Поэтому, в случае успешного доказательства УК сообщением от SMT-решателя является "unsat", что означает невыполни-

мость.

Но SMT-решатели не удовлетворяют другой задачи в проекте C-lightVer, а именно автоматизации доказательства УК, содержащих применение рекурсивных функций *rep*. Отметим, что изначально методы доказательства по индукции не являлись приоритетом при разработке SMT-решателей. В настоящее время для некоторых SMT-решателей, таких как Z3, CVC4, предложены тактики для доказательства по индукции. Но, в экспериментах, проведенных в C-lightVer, SMT-решатель Z3 не справился с доказательством ни одного УК, содержащего применение функции *rep*, несмотря на применение стратегии, предложенной Лейно [144]. Аналогичными оказались результаты применения системы Simplify. Лучше оказались результаты применения SMT-решателя CVC4, который справлялся с доказательством УК в случае экспериментов, представляющих финитные итерации над неизменяемыми массивами с выходом из цикла [144]. Но в других экспериментах SMT-решатель CVC4 не справился с доказательством ни одного УК программ с финитными итерациями над изменяемыми массивами с выходом из цикла [117, 144]. Также ни один из применяемых в C-lightVer SMT-решателей не смог сгенерировать контрпример в случае экспериментов по локализации ошибок в программах, УК которых содержат применение функции *rep*. Поэтому, в настоящее время SMT-решатели являются вспомогательными системами доказательства в системе C-lightVer.

Рассмотрим другой класс систем доказательства теорем, класс интерактивных систем. Основным преимуществом таких систем являются большие наборы готовых теорий для различных предметных областей, что важно для верификации различных программ. Другим преимуществом являются большие наборы тактик доказательства. Отметим, что применение готовых тактик иногда помогает существенно упростить доказательство. Основным недостатком таких систем является интерактивный характер доказательства, что приводит к сложностям с автоматизацией этого процесса. Этот недостаток можно частично компенсировать применением сложных тактик, существенно упрощающих доказываемую формулу. Иногда это позволяет сразу доказать УК.

В C-lightVer применялась интерактивная система доказательства PVS, под-

держивающая конструирование новых тактик для автоматизации доказательства. Новые тактики можно конструировать из команд доказательства и из существующих тактик. В системе C-lightVer применяется тактика induct-and-simplify, аргументом которой является имя переменной, являющейся параметром индукции. Эта тактика запускает доказательство по индукции, пытается упростить базу индукции и индукционный шаг. В экспериментах, проведенных в C-lightVer, эта тактика позволяла доказывать УК программ с финитными итерациями над неизменяемыми массивами без выхода из цикла [114]. Но в случае изменяемых массивов или выхода из цикла данная тактика не справилась с доказательством ни одного УК [114, 117]. Поэтому, PVS в настоящее время тоже не является основной системой доказательства в системе C-lightVer.

Промежуточное положение между автоматизированными и интерактивными системами доказательства занимает ACL2 [98, 110, 150]. С одной стороны система ACL2 автоматически пытается доказать все формулы в поданной на вход теории. С другой стороны, в случае неудачи доказательства какой-либо формулы, пользователь сам должен дополнить теорию нужной леммой. В настоящее время система ACL2 является единственной широко распространенной системой доказательства, ядро которой было формально верифицировано [69]. Корректность ядра системы ACL2 была доказана в рамках проекта Milawa [69]. Отметим, что логика системы ACL2 основана на вычислимых рекурсивных функциях [109]. Система ACL2 ориентирована на доказательство формул, содержащих применение рекурсивных функций. Таким образом, система ACL2 удовлетворяет и приоритету автоматизации, и приоритету корректности, и приоритету поддержки автоматического доказательства по индукции. Поэтому, в настоящее время ACL2 является основной системой доказательства в системе C-lightVer [117]. Рассмотрим систему ACL2 подробнее.

1.8.2. Входной язык системы ACL2

Входным языком системы ACL2 [98, 109, 110, 150] является аппликативный диалект языка Common Lisp (Applicative Common Lisp), в котором поддержи-

вается только функциональная парадигма и отсутствует поддержка императивной. Далее под языком ACL2 будем понимать входной язык системы ACL2. В дальнейших экспериментах будем записывать формулы, используя синтаксис языка системы ACL2, поэтому, приведем краткий обзор данного языка.

Язык Common Lisp ориентирован на обработку списков. Рассмотрим используемые в языке ACL2 операции над списками, позволяющие моделировать соответствующие операции над массивами. Это две операции для обработки индексированных последовательностей. Рассмотрим функции nth и *update-nth*, реализующие эти операции. Если i — индекс, l — список, то $(nth\ i\ l)$ — значение i-го элемента списка l. Если *expr* — выражение языка ACL2, то *(update-nth i expr l)* — новый список, который совпадает со списком l за исключением i-го элемента, значением которого является значение *expr*.

Рассмотрим другие используемые в языке ACL2 операции над списками. Предикат *integer-listp* истинен тогда и только тогда, когда аргументом является список и все его элементы имеют целочисленный тип. Предикат *equal* истинен тогда и только тогда, когда равны и длины списков, и их элементы. Функция $length$ вычисляет длину списка.

Моделировать новые типы данных позволяет библиотека fty языка ACL2, в которой работа с типизированными переменными сводится к использованию специальных функций. Каждому типу соответствует предикат, определяющий принадлежность к данному типу, функция приведения к данному типу и отношение эквивалентности. Такие типы можно задать с помощью специальных конструкций библиотеки fty. Макрос *fty::defprod* задает тип, аналогичный инструкции **struct** языка C-light. Если *st* — такой тип, то будут сгенерированы связанные с типом *st* функции с названиями *st-p*, *st-fix* и *st-equiv* соответственно. Также с помощью *fty::defprod* будет сгенерирован конструктор, макросы *make-st, change-st* и функции доступа к значениям полей. Пусть fd — поле структуры s, имеющей тип st. Тогда *(change-st s :fd expr)* — новая структура типа st, совпадающая со структурой s за исключением поля fd, значением которого является значение $expr$. Если fd — единственное поле структуры st, то *(make-st :fd expr)* — новая структура типа st, у которой значением поля fd

является значение $expr$. Тогда *(st->fd s)* значение поля fd структуры s. Другим способом доступа к данному значению является $s.fd$.

Макрос $b*$ позволяет промоделировать последовательное исполнение инструкций. Он является расширением макроса $let*$ языка ACL2, позволяющего удобным образом задать вложенный let. Рассмотрим общий вид конструкции $let*$: (*let** ((*var*$_1$ *term*$_1$) ... (*var*$_n$ *term*$_n$)) *body*), где все var_i являются переменными (не обязательно различными), *body* и все $term_i$ являются выражениями языка ACL2. Эта конструкция эквивалентна следующей:

$$(let\ ((var_1\ term_1))\ldots(let\ ((var_n\ term_n))\ body)\ \ldots)$$

Таким образом, связывание переменных val_i со значениями соответствующих выражений $term_i$ происходит последовательно. Значением такой конструкции является значение выражения *body*. Отметим, что каждая пара (var_i $term_i$) называется связыванием переменной var_i и значения выражения $term_i$, а переменная var_i называется локальной переменной в блоке $let*$.

Рассмотрим общий вид конструкции $b*$:

(b <list-of-bindings>.<list-of-result-forms>)*

где *<list-of-bindings>* — список конструкций *binding*, *<list-of-result-forms>* — список выражений языка ACL2. Значением конструкции $b*$ является последнее выражение списка *<list-of-result-forms>*. По аналогии с конструкцией $let*$ операции *binding* выполняются последовательно. При этом, общим видом конструкции *binding* является *(<binder-form> [<expression>])*,
где *<binder-form>* — конструкция *b*-binder*, *<expression>* — выражение языка ACL2. Так как $b*$ является расширением $let*$, то частным случаем конструкции *binding* является связывание переменной и значения выражения. При этом, переменная становится локальной переменной блока $b*$.

Таким образом, одним из возможных видов конструкции *<binder-form>* является переменная. Другим возможным видом конструкции *<binder-form>* является (*when condition*), где *condition* — логическое выражение языка ACL2. Пусть блок $b*$ содержит *binding*-конструкцию ((*when condition*) *expression*).

Если выражение *condition* истинно, то следующие за данной конструкцией операции *binding* не исполняются, а значением блока $b*$ является *expression*.

Язык ACL2 поддерживает многозначные функции. Конструкция *mv* позволяет функции возвратить более одного значения. Для связывания многих переменных с результатом исполнения многозначных функций используется следующая конструкция *binding*:

$$((mv\ x_1\ x_2\ ...\ x_{n-1}\ x_n)\ (form\text{-}returning\text{-}n\text{-}values))$$

где x_i — переменная для каждого i ($1 \leq i \leq n$), *(form-returning-n-values)* — вызов функции, возвращающий n значений. Таким образом i-я переменная x связывется с i-м значением *(form-returning-n-values)*.

Язык ACL2 обеспечивает поддержку пользовательских теорий. Формулы задаются с помощью логических связок *not* (логическое «не»), *or* (логическое «или»), *and* (логическое «и»), *implies* (импликация). В таких логических выражениях удобно использовать предикаты для проверки принадлежности переменной определенному типу. Например, предикат *integerp* проверяет принадлежность аргумента целочисленному типу. Конструкция $define$ задает функцию на языке ACL2, а также позволяет вводить леммы о ней. С помощью конструкции $defrule$ задаются леммы, теоремы и специальные секции, содержащие информацию, которая помогает ACL2 доказать теорему. Например, в секции *prep-lemmas* можно задать леммы, позволяющие доказать определяемую конструкцией $defrule$ теорему. Секция *enable* позволяет при доказательстве использовать леммы о функциях, определенных в $define$. Секция *induct* разрешает проведение доказательства по индукции. Аргументом данной секции служит применение специальной функции к переменным теоремы. Рекурсивное определение такой функции задает схему индукции. Например, рекурсивное определение библиотечной функции *dec-induct* определяет классическую схему индукции по натуральному параметру с шагом 1.

Язык системы ACL2 применяется в системе C-lightVer в качестве языка спецификаций [117] наравне с языком ACSL.

1.8.3. Система доказательства ACL2

Основным преимуществом системы ACL2 [98, 109, 110, 150] являются автоматические подстановки, основанные на специальных правилах вывода. Эти правила вывода называются правилами класса rewrite. Система ACL2 генерирует такие правила, используя теоремы класса rewrite. В дальнейших экспериментах система ACL2 автоматически доказала некоторые формулы, поэтому опишем возможности ACL2, которые позволяют автоматизировать доказательство.

Рассмотрим общий вид теоремы класса rewrite. Для записи теоремы класса rewrite в общем виде будем использовать язык Applicative Common Lisp. Так как язык Applicative Common Lisp является диалектом языка Common Lisp, то будем использовать префиксную нотацию. Теорема класса rewrite (без конструкций для задания имени теоремы) имеет следующий вид:

$$(implies\ (and\ h_1\ \ldots\ h_n)\ (equiv\ lhs\ rhs))$$

где

1. $implies$ является импликацией;
2. and является конъюнкцией;
3. $equiv$ является отношением эквивалентности (либо равенство $equal$ либо эквивалентность iff);
4. $(and\ h_1 \ldots h_n)$ является условием применения теоремы;
5. $h_1\ \ldots\ h_n$ являются гипотезами;
6. lhs является шаблоном;
7. rhs является замещающим выражением.

По умолчанию, переменные в таких теоремах являются переменными под неявными кванторами всеобщности. Переменные под кванторами существования или под вложенными кванторами необходимо задавать явно с помощью специальных конструкций.

То есть, теоремы класса rewrite имеют вид импликаций, заключением которых является равенство или эквивалентность. Посылка такой импликации называется условием применения. Левая часть такого равенства или эквивалентности называется шаблоном, правая часть такого равенства или эквивалентности называется замещающим выражением.

Система ACL2 генерирует правило класса rewrite для каждой теоремы класса rewrite. Структура таких правил совпадает со структурой теоремы. Названия таких правил также совпадают с названиями теорем. Система ACL2 может автоматически применять полученные правила вывода.

Система ACL2 применяет правила класса rewrite для доказательства других теорем. Это применение состоит в следующей подстановке: все выражения доказываемой теоремы, которые можно сопоставить с шаблоном какого-либо правила, заменяются на замещающее выражение этого правила в случае успешного сопоставления посылки доказываемой теоремы и условия применения. То есть, выполняются два алгоритма сопоставления. Первый алгоритм состоит в сопоставлении выражения доказываемой теоремы и шаблона правила. Выражения доказываемой теоремы сопоставляются с шаблонами правил в порядке, обратном заданию теорем, соответствующих правилам. Если первый алгоритм сообщает об успешном сопоставлении, то применяется второй алгоритм. Второй алгоритм состоит в сопоставлении посылки доказываемой теоремы и условия применения правила. Сначала второй алгоритм пытается выполнить подстановку, найденную первым алгоритмом, в посылке доказываемой теоремы и в условии применения. Потом второй алгоритм пытается сопоставить все конъюнкты условия применения с отдельными конъюнктами доказываемой теоремы. Если второй алгоритм сообщает об успешном сопоставлении, то к замещающему выражению применяется подстановка, найденная первым алгоритмом, и выражение доказываемой теоремы заменяется на полученное замещающее выражение.

Таким образом, правила класса rewrite задают систему подстановок. В отличие от классических подстановок, эти подстановки запускаются в случае успешного сопоставления шаблона и успешного сопоставления условия применения. Использование этих подстановок позволяет автоматизировать доказатель-

ство во многих случаях.

В случае применения в доказываемой теореме рекурсивной функции, система ACL2 использует доказательство по индукции, используя в качестве схемы индукции определение этой рекурсивной функции. В случае применения в доказываемой теореме нескольких рекурсивных функций, система ACL2 может перебирать схемы индукции, соответствующие определениям этих рекурсивных функций. Автоматическое применение схем индукции позволяет автоматизировать доказательство во многих случаях.

Автоматическое применение правил класса rewrite и автоматическое применение схем индукции является преимуществом системы ACL2.

1.9. Модуль трансляции конструкций C-kernel в конструкции C-light

Процедура, позволяющая от конструкций промежуточной C-kernel программы вернуться к конструкциям исходной C-light программы, работает итеративно [8]. На каждой итерации происходит просмотр информации обо всех применениях правил трансляции, то есть всей дополнительной информации, добавленной транслятором. При этом просмотре происходит поиск дополнительной информации, в котором содержится наибольший номер применения правила трансляции, т.е. той, в которой поле count имеет наибольшее значение. Далее, происходит извлечение информации о том, к применению какого правила трансляции относится данная дополнительная информация. То есть, происходит просмотр поля id, записанного в обрабатываемой дополнительной информации. Потом происходит применение правила трансляции, обратного к рассматриваемому. Рассматриваемая дополнительная информация удаляется. Записывается новая дополнительная информация, которая имеет следующую структуру:

- перед участком кода, получившимся в результате применения обратного правила трансляции, пишется C-комментарий, содержащий следующую

информацию:

1. `begin reverse` — информация, указывающая на то, что перед этим участком кода было применено обратное правило трансляции;

2. `range` – диапазон номеров строк, взятый из удаленной дополнительной информации.

- после участка кода, получившегося в результате применения обратного правила, пишется С-комментарий, содержащий слова `end reverse`.

Рассмотрим пример исполнения одной из итераций. Пусть на данной итерации в дополнительной информации, записанной в нижеприведенном участке кода, поле `count` имеет самое большое значение.

```
56. /* begin changes Norm6 17 70-74 */ int i = k++
57. if (i)
58. {
59.    j++;
60. } /* end changes */
```

Числа `56-60` — номера строк, добавленные для наглядности. Здесь

- `/* begin changes Norm6 17 70-74 */` и `/* end changes */` – комментарии, содержащие дополнительную информацию;

- `BCE5` — это id, уникальный идентификатор данного правила трансляции;

- `17` — это count, номер применения правила трансляции;

- `70-74` — это interval, то есть диапазон номеров измененных строк.

После исполнения итерации вышеприведенный участок кода заменится на:

```
56. /* begin reverse 70-74 */ if (k++)
57. {
58.  j++;
59. } /* end reverse */
```

Числа `56-59` — номера строк, добавленные для наглядности. Здесь

- `/* begin reverse 70-74 */` и `/* end reverse */` — комментарии, содержащие дополнительную информацию;
- `70-74` — это `range`, то есть диапазон номеров строк.

Последней итерацией будет являться та итерация, в ходе которой будет рассмотрена дополнительная информация, значение поле `count` которой равно `1`. При обнаружении ошибок, возникающих при верификации, процедура получает данные о диапазоне строк аннотированной C-kernel программы, в которых эту ошибку следует искать. Далее, процедура исполняет все итерации и в результате их исполнения получает исходную аннотированную C-light программу с добавленной им дополнительной информацией. В этой дополнительной информации процедура отыскивает самый узкий диапазон, в который вкладывается рассматриваемый диапазон строк аннотированной C-kernel программы. Дополнительная информация, в которой был найден искомый диапазон, ограничивает некоторый участок программы с помощью комментария `/* begin reverse ... */` и соответствующего комментария `/* end reverse */`. Этот участок и содержит ошибку.

1.10. Исходная версия системы C-lightVer

Одним из результатов диссертации стала модифицированная версия системы верификации C-lightVer, описанная в разделе 4.1. Для сравнения приведем описание исходной версии системы C-lightVer. Система верификации C-lightVer базируется на классическом дедуктивном подходе Хоара [38]. Схема системы C-lightVer изображена на рис. 1.1.

Система состоит из четырех основных модулей:

1. Модуль трансляции из C-light в C-kernel принимает на вход аннотированную C-light программу и производит
    - трансляцию в эквивалентную аннотированную C-kernel программу;

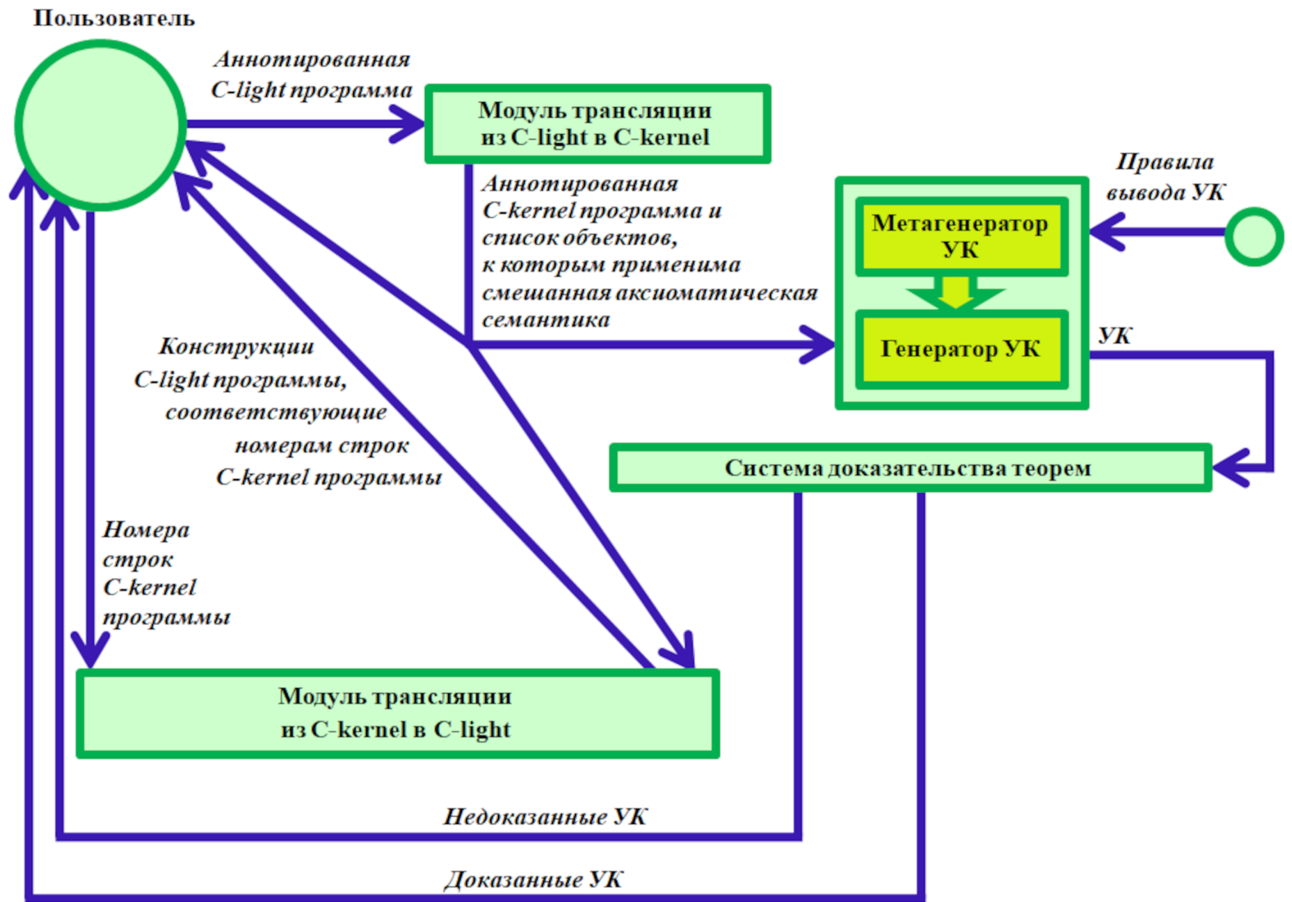

Рис. 1.1. Исходная версия системы C-lightVer

- поиск объектов, к которым применима смешанная аксиоматическая семантика;
- добавление информации [147], позволяющей транслировать конструкции C-kernel программы в конструкции C-light программы.

Эта информация добавляется к конструкциям, измененным в результате трансляции. Это позволяет снабдить конструкции названиями примененных правил трансляции.

2. Модуль метагенерации УК принимает на вход аннотированное промежуточное представление программы и правила вывода на языке задания шаблонов. Данный модуль порождает формулы, из истинности которых следует корректность программы.

3. Модуль доказательства теорем проверяет истинность УК. В качестве данного модуля в системе C-lightVer используются системы Simplify, Z3, CVC4,

PVS и ACL2. Если все УК доказаны, то C-lightVer завершает исполнение.

4. Модуль трансляции конструкций C-kernel в конструкции C-light основан на использовании информации, добавленной транслятором из C-light в C-kernel. Эта информация позволяет применить специальные правила трансляции [147] из C-kernel в C-light. Пользователю передаются недоказанные УК и промежуточная программа на языке C-kernel. Пользователь может анализировать промежуточную C-kernel программу, используя недоказанные УК. Если в результате такого анализа найдена ошибка в C-kernel программе, то пользователь может запрашивать данный модуль о соответствии промежуточной и исходной программы, передав номера строк промежуточной программы. Если от пользователя поступает запрос о соответствии промежуточной и исходной программы, то данный модуль применяет обратные правила трансляции и ищет код исходной программы, соответствующий номерам строк промежуточной программы. В отчете о соответствии промежуточной и исходной программы пользователю сообщается найденный код исходной программы.

Модули пронумерованы в соответствии с этапами исполнения исходной версии системы C-lightVer.

1.11. Выводы

На основе сделанного описания методов дедуктивной верификации и исходной версии системы C-lightVer можно сделать следующие выводы:

- В виду алгоритмической неразрешимости общей постановки задачи дедуктивной верификации, интерес представляют проблемно-ориентированные алгоритмы и методы. Для применения дедуктивной верификации на практике представляют интерес определенные классы программ, для которых можно избежать задания инвариантов циклов, например, финитные итерации над последовательностями данных.

- Разработанный ранее алгоритм генерации операции замены поддерживает только такой класс финитных итераций, как итерации над неизменяемыми структурами данных без / с выходом из цикла и без / с условными инструкциями, не вложенными друг в друга. Поэтому интерес представляет разработка алгоритмов генерации операции замены для таких классов финитных итераций, как итерации над изменяемыми структурами данных без / с выходом из цикла и без / с вложенными условными инструкциями.

- Применение символического метода верификации финитных итераций для данного класса циклов приводит к появлению в условиях корректности рекурсивной функции rep. Но в методе локализации ошибок Денни и Фишера предлагается ограниченный набор семантических меток, который не включает метки для применения рекурсивных функций, выражающих результат тела цикла. Это приводит к невозможности генерировать подробные объяснения для определения функции rep, соответствующего телу цикла. Поэтому интерес представляет расширение набора семантических меток специальными метками для финитных итераций над последовательностями данных. Также интерес представляют алгоритмы генерации операции замены, снабжающие такими семантическими метками определение финитных итераций. Кроме того, представляет интерес разработка эвристических методов, позволяющих упростить и автоматизировать доказательство условий корректности, содержащих применение рекурсивных функций rep.

- В исходной версии системы C-lightVer заложены возможности по расширяемости с помощью языка шаблонов, разработанного для поддержи метагенерации УК. Однако, для расширения системы C-lightVer такими конструкциями, как финитные итерации над последовательностями данных, языка шаблонов недостаточно, так как требуется реализовать в системе C-lightVer алгоритмы генерации рекурсивных функций, выражающих результаты различных видов финитных итераций. Поэтому, актуальной является задача реализации таких алгоритмов в системе C-lightVer.

- Если система доказательства теорем не справляется с доказательством УК, то в исходной версии системы C-lightVer недоказанные УК передаются пользователю для анализа. Во-первых, важной является задача не полагаться только на возможности систем доказательства, а разработать набор стратегий доказательства УК. Так как такие стратегии могут использовать информацию о структуре исходной программы, структуре УК, то применение таких стратегий может позволить системе доказательства установить истинность таких УК, которые эта система не доказала бы без этих стратегий. Во-вторых, если применение стратегий не привело к доказательству УК, то актуальной является задача предоставления пользователю текста на естественном языке о соответствии УК и фрагментов программы. Также в случае недоказанных УК важной задачей является разработка стратегий, которые могут предупреждать о возможном наличии ошибок в конкретных программных конструкциях.

В первой главе диссертации были обозначены актуальные проблемы, возникающие при дедуктивной верификации. В следующих главах рассмотрим результаты, представленные в данной диссертации для решения данных проблем в случае верификации программ с финитными итерациями.

Глава 2

Генерация условий корректности программ с финитными итерациями и стратегии автоматизации их доказательства

В данной главе описаны алгоритмы генерации функций, выражающих результаты различных классов финитных итераций [117, 121, 144]. Использование таких алгоритмов приводит к генерации УК, содержащих применения функции rep. В данной главе описаны также стратегии доказательства таких УК [117, 118, 121, 144]. Для стратегии усиления УК сформулирована и доказана теорема о корректности [123, 125].

2.1. Алгоритм генерации функций, выражающих результаты итераций с вложенными условными инструкциями над неизменяемыми массивами

Опишем алгоритм генерации функций, выражающих результаты итераций с вложенными условными инструкциями над неизменяемыми массивами [144] как модификацию алгоритма из раздела 1.4.3. Для ослабления ограничений применимости алгоритма генерации операции замены была разработана его модификация. Опишем отличия модифицированного алгоритма [144] от исходного [146]. Во-первых, был предложен способ генерации операции замены для вложенных инструкций if. Для каждой вложенной инструкции **if** вида

$$\mathbf{if}\,(\mathbf{e_1}(\mathbf{i},\mathbf{x_1},\mathbf{x_2},\ldots,\mathbf{x_k}))\{\mathbf{A_1};\mathbf{if}\,(\mathbf{e_2}(\mathbf{i},\mathbf{x_1},\mathbf{x_2},\ \ldots,\mathbf{x_k}))\mathbf{A_2}\,\mathbf{else}\,\mathbf{A_3};\mathbf{A_4}\}\,\mathbf{else}\,\{\mathbf{B};\},$$

где $\mathbf{A_1}, \mathbf{A_2}, \ldots \mathbf{A_4}$ являются составными выражениями, содержащими операции присваивания, генерируются следующие аксиомы:

$$\forall x_1 \forall x_2 \dots \forall x_k \ ((e_1(i, x_1, x_2, \dots, x_k) \Rightarrow A_1^*) \wedge e_2(i, x_1, x_2, \dots, x_k)) \Rightarrow (A_2; A_4)^*$$
$$\forall x_1 \forall x_2 \dots \forall x_k \ ((e_1(i, x_1, x_2, \dots, x_k) \Rightarrow A_1^*) \wedge \neg e_2(i, x_1, x_2, \dots, x_k)) \Rightarrow (A_3; A_4)^*$$
$$\forall x_1 \forall x_2 \dots \forall x_k \ \neg e_1(i, x_1, x_2, \dots, x_k) \Rightarrow B^*$$

где A_1^*, $(A_2; A_4)^*$, $(A_3; A_4)^*$ и B^* являются результатами применения к A_1, $(A_2; A_4)$, $(A_3; A_4)$ и B соответственно специальной процедуры из раздела 1.4.3. Для множественных вложенных инструкций **if** аксиомы генерируются тем же способом. Во-вторых, для каждого $m(1 \leq m \leq k)$ автоматически генерируется rep_m для задания операции замены каждой переменной цикла x_m. Для каждого $m(1 \leq m \leq k)$ rep_m соответствует значению переменной x_m. Поэтому, для каждого $m(1 \leq m \leq k)$ генерация rep_m основана на замене m-й компоненты rep на rep_m. В системе C-lightVer реализация этого алгоритма позволяет генерировать код [114, 144] на входном языке систем Simplify, Z3, CVC4 и PVS.

2.2. Алгоритм генерации функций на языке системы ACL2, выражающих результаты итераций над изменяемыми массивами языка C-kernel

Так как предыдущий алгоритм ориентирован только на неизменяемые массивы, то был предложен новый алгоритм. Опишем алгоритм генерации функций на языке системы ACL2, выражающих результаты итераций над изменяемыми массивами языка C-kernel. Данный алгоритм [117] состоит из двух шагов:

1. Проверка итерации на применимость к ней данного алгоритма и генерация вектора v изменяемых переменных цикла.

2. Генерация определения функции rep с помощью трансляции тела цикла на язык системы ACL2.

Рассмотрим первый шаг алгоритма.

2.2.1. Класс итераций, для которых применяется алгоритм генерации функций, выражающих результаты итераций над изменяемыми массивами

Пусть S — одномерный массив из n элементов простого типа языка C-light и $S \in v$. Рассмотрим специальный случай финитной итерации над массивом S:

$$\mathbf{for}\,(\mathbf{i} = \mathbf{0}; \mathbf{i} < \mathbf{n}; \mathbf{i}++)\ \mathbf{v} := \mathbf{body}(\mathbf{v}, \mathbf{i})\ \mathbf{end},$$

где тело цикла является допустимой конструкцией [117].

Допустимая конструкция — это один из следующих операторов языка C-kernel:

1. Пустой оператор, в том числе пустой блок.

2. Оператор выхода из цикла **break**;.

3. Оператор присваивания $\mathbf{a} = \mathbf{b}$;, где a — переменная простого типа, либо имеет вид $S[i]$, b — выражение языка C-kernel.

4. Условный оператор **if** $(\mathbf{a})$ **b**, где a — выражение языка C-kernel, b — допустимая конструкция.

5. Условный оператор **if** $(\mathbf{a})$ **b else c**, где a — выражение языка C-kernel, b и c — допустимые конструкции.

6. Блок $\{\mathbf{a_1}\ \mathbf{a_2}\ \ldots\ \mathbf{a_{k-1}}\ \mathbf{a_k}\}$, где a_r — допустимая конструкция для каждого r: $1 \leq r \leq k$.

7. Вложенная финитная итерация $\mathbf{for}\,(\mathbf{j} = \mathbf{0}; \mathbf{j} < \mathbf{m}; \mathbf{j}++)\ \mathbf{u} := \mathbf{body}(\mathbf{u}, \mathbf{j})\ \mathbf{end}$, где u — вектор переменных, изменяемых вложенной финитной итерацией.

В вектор v входит массив S и переменные простого типа, которые могут изменяться в теле цикла. Введем функцию w, ее аргументом является допустимая конструкция и она возвращает множество элементов вектора v для данной

конструкции. Определим такую функцию для каждого вида допустимой конструкции op:

1. Если op — это пустой оператор (в том числе пустой блок), то $w(op) = \emptyset$.

2. Если op — это **break;**, то $w(op) = \emptyset$.

3. Если op – это **a = b;**, где a — переменная простого типа либо имеет вид $S[i]$, b — выражение языка C-kernel, то возможны два варианта:

   а. Если a — переменная простого типа, то $w(op) = \{a\}$.

   б. Если a имеет вид $S[i]$, то $w(op) = \{S\}$.

4. Если op — это **if (a) b**, то $w(op) = w(b)$.

5. Если op — это **if (a) b else c**, то $w(op) = w(b) \cup w(c)$.

6. Если op — это $\{\mathbf{a_1\ a_2\ \ldots\ a_{k-1}\ a_k}\}$, то $w(op) = w(a_1) \cup w(a_2) \cup ... \cup w(a_{k-1}) \cup w(a_k)$.

7. Если op — это $\mathbf{for\,(j = 0; j < m; j++)\ u := body(u, j)\ end}$, то $w(op) = w(body)$.

Пусть T — результат применения функции w к телу цикла. Рассмотрим произвольное упорядочивание элементов множества T. Обозначим его как вектор v. Это вектор изменяемых переменных цикла, который используется при трансляции тела цикла на язык системы ACL2.

2.2.2. Трансляция тела цикла на язык системы ACL2

Рассмотрим генерацию определения функции rep с помощью трансляции тела цикла на язык системы ACL2. Генерация определения функции rep состоит из трех этапов [117]. На первом шаге осуществляется генерация типа данных $frame$ с помощью конструкции *fty::defprod*. Этот тип данных представляет

собой структуру, полями которой являются элементы вектора v, а также булевское поле **loop-break**. Также конструкция *fty::defprod* генерирует макросы *make-frame* и *change-frame*, функции *frame-p*, *frame-fix*, *frame-equiv* и функции доступа к значениям полей.

Значения полей переменной типа $frame$ являются значениями соответствующих переменных в ходе исполнения цикла. Поле **loop-break** истинно тогда и только тогда, когда сработал оператор **break**. Таким образом, переменная типа $frame$ хранит значения вектора v в ходе исполнения цикла. Поэтому необходим способ задания начальных значений элементов вектора v, то есть значений перед исполнением цикла. Для этого на втором этапе с помощью конструкции *make-frame* происходит генерация функции *frame-init*. Эта функция возвращает структуру типа $frame$ с заданными начальными значениями полей. Значением поля **loop-break** у такой структуры является nil. Это означает, что перед исполнением цикла оператор **break** не срабатывает.

На третьем этапе происходит генерация функции rep по телу цикла следующим образом. Будем считать, что нулевая итерация цикла соответствует значениям элементов вектора v перед исполнением цикла. Поэтому

$$rep(0, \textit{frame-init}\ (v_0)) = \textit{frame-init}\ (v_0)$$

Это ограничивает глубину рекурсии.

В случае выхода из цикла на итерации l $(0 < l \leq n)$ при исполнении оператора **break** будем считать, что итерации цикла продолжаются, но значения v не изменяются на таких итерациях j, что $l \leq j \leq n$. Зададим rep так, что

$$rep(l, \textit{frame-init}\ (v_0)) = rep(j, \textit{frame-init}\ (v_0))$$

Поэтому необходимо проверять на истинность значение поля **loop-break** у возвращенной на предыдущей итерации структуры типа $frame$. В случае, когда выход из цикла происходит на текущей итерации, необходимо прекратить исполнение и вернуть такую структуру типа $frame$, что ее поле **loop-break** будет истинным.

Переменную структурного типа $frame$, поля которой соответствуют переменным вектора v, будем обозначать как fr. При необходимости изменить значения полей переменной fr создается новая переменная fr с новыми значениями полей. Это позволяет удобным образом обрабатывать элементы вектора v.

Алгоритм генерации функции *rep* состоит из двух шагов. На первом шаге используется функция *generate_rep_one*, которая строит сигнатуру функции *rep* и конструкцию $b*$. Эта конструкция $b*$ является самым верхним по уровню вложенности блоком кода функции *rep* на языке ACL2. Этот блок соответствует составному оператору, являющемуся телом цикла. Такая конструкция позволяет промоделировать последовательное исполнение следующих инструкций: проверку условия выхода из рекурсии с помощью конструкции *when*, рекурсивный вызов *rep* для предыдущей итерации, результат которого сохраняется в переменной fr, проверку с помощью конструкции *when* выхода из цикла на предыдущей итерации.

Каждая такая инструкция блока $b*$ является конструкцией *binding*. Следующие за ними в этом блоке $b*$ инструкции *binding* моделируют последовательное исполнение тела цикла, совершая операции над структурой fr. Они будут сгенерированы функцией *generate_rep* на втором шаге алгоритма. Кроме того, на первом шаге алгоритма функция *generate_rep_one* генерирует выражение fr, являющееся значением данного блока $b*$. Таким образом, если не было выхода из цикла, то значением функции *rep* на текущей итерации будет переменная fr, полученная при применении к результату предыдущей итерации конструкций, соответствующих телу цикла.

На втором шаге алгоритма используется функция *generate_rep*, генерирующая конструкции, соответствующие телу цикла. Отметим, что для любой финитной итерации данного вида на первом шаге алгоритма генерируется один и тот же код на языке ACL2. То есть различия между разными финитными итерациями учитываются не на первом, а на втором шаге алгоритма.

Функция *generate_rep* осуществляет трансляцию тела цикла на язык системы ACL2. Далее под объемлющим блоком для конструкции *binding* будем по-

нимать блок $b*$, в котором она находится. Функция $c2acl2$ осуществляет трансляцию C-kernel выражения на язык ACL2. Определим функцию $generate_rep$ для каждого вида допустимой конструкции op:

1. Если op – это пустой оператор (в том числе пустой блок), то результатом $generate_rep(op)$ будет $(fr\ \ fr)$. Данная запись является конструкцией *binding* в объемлющем блоке $b*$. Она означает, что внутри блока создается новая переменная fr со значением прежней и с новой областью видимости. Т. е. пустой оператор (в том числе пустой блок) не меняет значения элементов вектора v.

2. Если op — это **break**;, то результатом $generate_rep(op)$ будет

   (fr (change-frame fr :loop-break t)) ((when t) fr)

   Данная запись представляет собой две последовательно идущие конструкции *binding* в объемлющем блоке $b*$. Первая конструкция *binding* означает, что внутри блока создается новая переменная fr со значением *(change-frame fr :loop-break t)* и с новой областью видимости. Т. е. у новой переменной fr поле *loop-break* стало истинным.

   Вторая конструкция *binding* означает, что при соблюдении условия $when$ происходит выход из объемлющего блока $b*$. Так как таким условием $when$ является t, то всегда происходит выход из объемлющего блока $b*$ при срабатывании данной второй конструкции. Т. е. при срабатывании оператора **break** происходит выход из цикла. Возвращаемым значением такого объемлющего блока $b*$ будет являться переменная fr, поле *loop-break* которой будет указывать на то, что необходимо также произвести выход из блоков, в которые вложен данный объемлющий блок (если они существуют).

3. Если op — это **a = b**;, где a — переменная простого типа либо имеет вид $S[i]$, b — выражение языка C-kernel, то возможны два варианта:

а. Если a — переменная простого типа, то результатом $generate_rep(op)$ будет

```
(fr  (change-frame  fr  :a  c2acl2(b)))
```

Данная запись является конструкцией *binding* в объемлющем блоке $b*$. Она означает, что внутри блока создается новая переменная fr со значением *(change-frame fr :a c2acl2(b))* и с новой областью видимости. Т. е. у новой переменной fr поле a принимает значение выражения b. Таким образом, оператор присваивания меняет значение переменной a вектора v.

б. Если a имеет вид $S[i]$, то результатом $generate_rep(op)$ будет

```
(fr  (change-frame  fr  :S (update-nth  i  c2acl2(b)
                    (frame->S  fr))))
```

Данная запись является конструкцией *binding* в объемлющем блоке $b*$. Она означает, что внутри блока создается новая переменная fr со значением

```
(change-frame  fr  :S (update-nth  i  c2acl2(b)
                    (frame->S  fr)))
```

и с новой областью видимости. Т. е. у новой переменной fr поле S принимает значение последовательности S, в которой на месте элемента с индексом i стоит значение выражения b. Таким образом оператор присваивания изменяет массив S из вектора v.

4. Если op — это **if** (**a**) **b**, то результатом $generate_rep(op)$ будет

```
(fr  (if  c2acl2(a) (b* (generate_rep(b)) fr) fr))
((when  (frame->loop-break  fr))  fr)
```

Данная запись представляет собой две последовательно идущие конструкции *binding* в объемлющем блоке $b*$. Первая конструкция *binding* означает, что внутри блока создается новая переменная fr с новой областью видимости и со значением

(if c2acl2(a) (b(generate_rep(b)) fr) fr)*

Т. е. при выполнении условия a значением переменной fr станет

(b (generate_rep(b)) fr)*

где $generate_rep(b)$ — результат трансляции b на входной язык системы ACL2. Отметим, что любой оператор b (в том числе составной) транслируется в одну или две *binding* конструкции. Блок $b*$ в ACL2-коде для оператора if является объемлющим для этих *binding* конструкций. Таким образом, при выполнении условия a значение вектора v будет определяться положительной ветвью условного оператора.

При невыполнении условия a значением переменной fr станет предыдущее значение fr. Таким образом, при невыполнении условия a значение вектора v не изменится.

Вторая конструкция *binding* означает, что при соблюдении условия $when$ происходит выход из объемлющего блока $b*$. Так как таким условием $when$ является *(frame->loop-break fr)*, то выход из объемлющего блока $b*$ происходит при истинности поля *loop-break* переменной fr. Т. е. если при исполнении оператора сработала инструкция $break$, то происходит выход из объемлющего блока $b*$. Возвращаемым значением такого объемлющего блока $b*$ будет являться переменная fr, поле $loop-break$ которой будет указывать на то, что необходимо также произвести выход из блоков, в которые вложен данный объемлющий блок (если они существуют).

5. Если op — это **if** (**a**) **b else c**, то результатом $generate_rep(op)$ будет

(fr (if c2acl2(a) (b(generate_rep(b)) fr) (b*(generate_rep(c)) fr)))*
((when (frame->loop-break fr)) fr)

Данная запись представляет собой две последовательно идущие конструкции *binding* в объемлющем блоке $b*$. Первая конструкция *binding* определяется по аналогии с пунктом 4.

Определение второй конструкции *binding* совпадает с определением из пункта 4.

6. Если op — это $\{\mathbf{a_1}\ \mathbf{a_2}\ ...\ \mathbf{a_{k-1}}\ \mathbf{a_k}\}$, то результатом $generate_rep(op)$ будет

$$(fr\ (b^*(generate_rep(a_1)\ generate_rep(a_2)\dots generate_rep(a_{k-1})$$
$$generate_rep(a_k))fr))((when\ (frame\text{-}{>}loop\text{-}break\ \ fr))\ \ fr)$$

Данная запись представляет собой две последовательно идущие конструкции *binding* в объемлющем блоке $b*$. Первая конструкция *binding* означает, что внутри блока создается новая переменная fr с новой областью видимости и со значением

$$(b^*(generate_rep(a_1)\ generate_rep(a_2)\dots$$
$$generate_rep(a_{k-1})\ generate_rep(a_k))fr)$$

где $generate_rep(a_r)$ для каждого r $(1 \le r \le k)$ — результат трансляции a_r на входной язык системы ACL2. Отметим, что любой оператор **c** (в том числе составной) транслируется в одну или две *binding* конструкции. Блок $b*$ в ACL2-коде для составного оператора является объемлющим для этих *binding* конструкций. Таким образом, значение переменной fr при исполнении конструкции $b*$ является значением вектора v при исполнении составного оператора.

Вторая конструкция *binding* определяется как в пункте 4.

7. Если op — это вложенная финитная итерация, то введем нумерацию финитных итераций в программе [121]. Для первой финитной итерации генерируется функция rep_1, для второй генерируется rep_2 и т.д. Расположение финитных итераций в программе можно представить как последовательность деревьев, корнями которых являются финитные итерации на самом верхнем уровне вложенности программы. Потомками любой вершины этих деревьев являются циклы, вложенные в цикл, соответствующий вершине. Сначала нумеруются итерации из первого дерева вложенности,

потом нумеруются итерации из второго дерева вложенности и т.д. Итерации внутри каждого дерева вложенности нумеруются путем обхода в ширину. При переходе к следующему дереву вложенности нумерация продолжается.

Определим трансляцию вложенных итераций на язык системы ACL2 с помощью функции $generate_rep$:

$$generate_rep(\mathbf{for}\,(\mathbf{j} = \mathbf{0}; \mathbf{j} < \mathbf{m}; \mathbf{j}++)\ \mathbf{u} := \mathbf{body}(\mathbf{u}, \mathbf{j})\ \mathbf{end}) =$$
$$(fr\ (rep_k\ m\ fr_k))$$

где

- k – номер финитной итерации.
- n – обозначение количества итераций.
- fr_k – обозначение вектора изменяемых переменных итерации i.

То есть вложенная финитная итерация транслируется в применение функции rep_k к аргументам m и fr_k. Далее запускается генерация определения функции rep_k с помощью описываемого алгоритма. Этот процесс завершится, так как число вложенных финитных итераций конечно.

Вопрос существования объемлющего блока решается построением дерева вложенности блоков для кода, сгенерированного на втором шаге алгоритма. В качестве корня такого дерева возьмем блок $b*$, сгенерированный функцией $generate_rep_one$ на первом шаге алгоритма. Этот блок соответствует телу цикла. Отметим, что каждому оператору тела цикла можно сопоставить блок в данном дереве вложенности блоков. Например, если if-ветвь представлена единственным оператором, то объемлющим для него будет блок $b*$, сгенерированный при трансляции данного if. А если оператор входит в последовательность инструкций составного оператора ветви if, то объемлющим для него будет блок $b*$, полученный при трансляции данного составного оператора. Поиск

объемлющего блока для определенного оператора в таком дереве вложенности можно проводить в глубину, пока не будет найден блок $b*$, одна из конструкций *binding* которого будет соответствовать данному оператору. Так как самый объемлющий блок — тело цикла, то такой поиск будет всегда завершаться.

2.2.3. Генерация функций, выражающих результат обратных финитных итераций

В ходе исследования рассматриваемый класс финитных итерации был расширен обратной итерацией:

$$\mathbf{for}\,(\mathbf{i} = \mathbf{n} - \mathbf{1}; \mathbf{i} >= \mathbf{0}; \mathbf{i} - -)\ \mathbf{v} := \mathbf{body}(\mathbf{v}, \mathbf{i})\ \mathbf{end},$$

где итерация осуществляется над n элементами.

Поэтому, в генератор функции *rep* были внесены изменения. Во-первых, генератор должен генерировать не только структуру $frame$, но и структуру $envir$. Структура *envir* хранит значения переменных, используемых в цикле, но не изменяемых в нем.

Также у структуры *envir* есть выделенное поле *upper-bound*. Значением поля *upper-bound* является недостижимая верхняя граница счетчика цикла, то есть $(n - 1) + 1$.

Во-вторых, в случае обратной итерации номер итерации не совпадает со значением счетчика цикла. Поэтому, генератор должен включать в поля структуры $frame$ не только элементы вектора v, но и счетчик цикла. Следовательно, появилась возможность использовать значение счетчика цикла после окончания исполнения цикла.

Но символический метод верификации финитных итераций основан на том, что счетчику цикла не могут быть присвоены произвольные значения. Поэтому, третье изменение генератора состоит в использовании в теле функции *rep* разности верхней границы и номера итерации в качестве значения счетчика цикла. В случае продолжения итерации счетчик цикла инициализируется разностью того же выражения и единицы. В случае выхода из цикла, значение

счетчика не изменяется. Отметим, что такой подход может упростить доказательство, так как выражает значение счетчика цикла в явном виде.

2.2.4. Расширение языка шаблонов и задание на нем правила вывода для финитных итераций

Разработка алгоритмов генерации операции замены позволила расширить систему C-lightVer правилом вывода для финитных итераций. Так как правило вывода для финитных итераций соответствует нормальной форме, то для упрощения расширения системы C-lightVer данным правилом вывода был применен метод метагенерации УК. Рассмотрим конструкции, которыми был расширен язык описания шаблонов для задания данного правила. Для описания конструкции $vector_substitution$ введем обозначения. Пусть вектор $v = \langle v_1 \dots v_k \rangle$ и для каждого $j (1 \leq j \leq k)$ выражение $expr_j$ является результатом замены всех вхождений терма $vector_element$ в выражении $expr$ на v_j. Тогда $vector_substitution(T,\ vec,\ expr)$ обозначает одновременную замену для каждого $j (1 \leq j \leq k)$ всех вхождений v_j в формуле T на выражение $expr_j$.

Рассмотрим правило вывода, предложенное для финитной итерации, на языке описания шаблонов [19, 114, 126]:

```
{P} prog {(vector_substitution(Q, v,
                 rep(n, v, S, body).vector_element)}
|- {any_predicate(P)} any_code(prog)
for(int_var(i) = 0; int_var(i) < int_var(n); int_var(i)++)
admissible_construct(i, n, v, S, body)
{any_predicate(Q)}
```

где конструкция $admissible_construct(i, n, v, S, body)$ соответствует допустимой конструкции тела цикла, int_var соответствует целочисленной переменной. Конструкция $vector_substitution(Q,\ v, rep(n,\ v,\ S,\ body).vector_element)$ обозначает одновременную замену для каждого $t (1 \leq t \leq length(v))$ всех вхождений v_t в формуле Q на $rep(n,\ v,\ S,\ body).v_t$. Правило вывода для обратной итерации определено подобным образом.

Реализация проверки цикла на соответствие конструкции *admissible_construct* описана как проверка принадлежности к классу итераций для которых применим алгоритм генерации *rep* в разделе 2.2.1. Если цикл был успешно сопоставлен данному шаблону, то применяется данное правило вывода и генерируется определение функции *rep* с помощью алгоритма, описанного в разделе 2.2.2.

2.3. Вспомогательные стратегии

Доказательство УК, содержащих операцию замены для финитной итерации, основано на индукции по длине массива, над которым осуществляется данная итерация. При этом, использования классической индукции в системе ACL2 недостаточно для доказательства УК, содержащих операцию замены для финитной итерации, в случае, если итерация изменяет элементы массива или содержит инструкцию `break`. Поэтому, были разработаны стратегии доказательства, основанные на структуре УК и на структуре финитной итерации, а также на индукции [117, 118, 121]. Среди этих стратегий можно выделить такие классы, как вспомогательные стратегии, основные стратегии и стратегии для классов финитных итераций. Вспомогательные стратегии отличаются тем, что целью их разработки было применение в составе других стратегий. Однако, стратегия для программ, спецификации которых содержат функции со свойством конкатенации, и стратегия для программ с финитными итерациями над массивами могут применяться отдельно.

2.3.1. Стратегия выбора посылок

Введем обозначения. Стратегия принимает в качестве аргумента финитную итерацию над массивом a длины n. Таким образом, в условии корректности вызов функции *rep* имеет следующие аргументы:

- Аргумент *iteration* соответствует номеру итерации. Общее количество

итераций равно $(n-1)+1$.

- Аргумент env соответствует вектору неизменяемых параметров цикла. Так как аргумент env является объектом типа $envir$, то $env.upper\text{-}bound$ – верхняя недостижимая граница цикла. В случае цикла, реализующего финитную итерацию, с максимальным значением счетчика $n-1$ это поле должно иметь значение $(n-1)+1$. При вызове функции rep в условии корректности в качестве аргумента используется выражение $envir\text{-}init((n-1)+1,...)$.

- Аргумент fr соответствует вектору v. Так как аргумент fr является объектом типа $frame$, то $fr.i$ – начальное значение счетчика итераций. Выражение $fr.loop\text{-}break$ – начальное значение индикатора выхода из цикла. Это поле всегда должно иметь значение nil, то есть ложь. Значением выражения $fr.a$ является массив до начала исполнения цикла. При вызове функции rep в условии корректности в качестве аргумента используется выражение $frame\text{-}init(n-1,...)$.

Вызов $rep((n-1)+1, fr, env)$ возвращает объект типа $frame$, соответствующий значению вектора v после итерации. Следовательно,

- Значением выражения $rep((n-1)+1, env, fr).i$ является значение счетчика цикла, полученное в результате итерации.

- Значением выражения $rep((n-1)+1, env, fr).loop\text{-}break$ является индикатор, сработал ли оператор `break`.

- Значением выражения $rep((n-1)+1, env, fr).a$ является массив, полученный в результате итерации.

Условием применимости стратегии выбора посылок является наличие финитной итерации (с возможным выходом из цикла) [118]. Данная стратегия применяется тогда, когда для доказательства условия корректности пытаются доказать утверждение о свойстве функции rep.

Обозначим такое утверждение как P. Таким образом, первым аргументом данной стратегии служит финитная итерация. Вторым аргументом этой стратегии является определение функции rep. Третьим аргументом данной стратегии является утверждение P.

Стратегия ориентирована на решение задачи о превращении утверждение P в лемму, имеющую вид импликации. Заключение такой импликации становится утверждение P. Тогда задача сводится к формированию посылки такой импликации. Такая посылка должна позволить применить такую лемму для доказательства условия корректности и также должна позволить доказать такую лемму. Простой вариант стратегии предполагает использовать в качестве посылки посылку условия корректности:

$$(X_1 \wedge ... \wedge X_n) \ ,$$

Обозначим такую посылку как T. Но в экспериментах, проведенных в C-lightVer, такая посылка не позволяла доказать истинность леммы в системах доказательства в автоматическом режиме. Основная проблема заключается в равенстве количества итераций и верхней недостижимой границе в посылке условия корректности. Если доказывать формулу с такой посылкой по индукции, то в индукционной гипотезе на единицу уменьшается не только количество итераций, но и недостижимая верхняя граница. Такая индукционная гипотеза бесполезна.

Поэтому, было принято решение использовать в качестве посылок стратегии более общие утверждения. В таких утверждениях не требуется равенства номера итерации и недостижимой верхней границы. В настоящее время стратегия основана на использовании в качестве посылки формулы

$$(iteration \in N) \wedge (iteration \leq env.upper\text{-}bound) \wedge$$
$$(env.upper\text{-}bound < (len fr.a)) \wedge (fr.j = (env.upper\text{-}bound - 1))$$

или

$$(env.upper\text{-}bound \in N) \wedge (env.upper\text{-}bound < (len fr.a)) \wedge$$
$$(fr.j = (env.upper\text{-}bound - 1)) \wedge \neg fr.loop\text{-}break$$

В дальнейшем планируется расширять набор посылок. Выбор из них определяется тем, какая из них помогает системе ACL2 доказать лемму. Это происходит следующим образом. Обозначаем рассматриваемую посылку как L и пытаемся доказать в системе ACL2 формулу $L \to P$. В случае успешного доказательства обозначим лемму $L \to P$ как R. Пусть E – подстановка $(n-1)+1$ вместо *iteration*, $(n-1)+1$ вместо *env.upper-bound*, $n-1$ вместо $fr.j$, *nil* вместо $fr.j$, a вместо $fr.a$. Обозначим как S результат применения подстановки E к формуле L и пытаемся доказать в системе ACL2 формулу $T \to S$. В случае успешного доказательства обозначим лемму $T \to S$ как Q. Обозначим как U результат применения подстановки E к формуле R и пытаемся с помощью лемм R и Q доказать в системе ACL2 формулу $T \to U$. В случае успешного доказательства обозначим лемму $T \to U$ как V и добавим леммы R, Q и V в теорию предметной области. Эти леммы могут помочь системе ACL2 доказать условие корректности.

2.3.2. Стратегия для программ, спецификации которых содержат функции со свойством конкатенации

Одним из важнейших способов упрощения доказательства является использование свойства конкатенации [121].

Определим такое свойство предикатов, как свойство конкатенации. Будем говорить, что предикат R обладает *свойством конкатенации*, если для R выполнено свойство вида

$$R(i, k, u_1, \dots, u_n) \wedge R(k+1, j, u_1, \dots, u_n) \to R(i, j, u_1, \dots, u_n)$$

Будем говорить, что предикат R обладает *свойством конкатенации со склейкой на границе по предикату* f, если для R выполнено свойство следующего вида:

$$(R(i, k, u_1, \dots, u_n) \wedge R(k+1, j, u_1, \dots, u_n) \wedge f(u_1[k], u_1[k+1]) \wedge \dots f(u_m[k], u_m[k+1])) \to R(i, j, u_1, \dots, u_n)$$

Для каждого $m(1 \leq m \leq n)$ будем называть выражение $f(u_m[k], u_m[k+1])$ условием склейки на границе для свойства конкатенации.

Пусть A – исходный массив, B – массив, полученный в результате применения функции rep к массиву A. Рассматриваемая стратегия генерирует утверждения о равенстве подмассивов массива A и подмассивов массива B. Эти подмассивы выбираются с помощью следующих эвристик для цикла for:

- Если счетчик цикла с инструкцией `break` возрастает (убывает), то генерируется утверждение, что равны подмассивы массивов B и A, начинающиеся с итогового значения счетчика (с нуля) до длины массива минус 1 (до итогового значения счетчика).

- Если i — счетчик цикла с выражениями вида $A[i + expr] = A[i]$, то генерируется утверждение, что подмассив массива B является результатом сдвига подмассива массива A.

- Если счетчик цикла возрастает (убывает) на произвольной итерации цикла, то генерируется утверждение, что равны подмассивы массивов B и A, начинающиеся с текущего значения счетчика (с нуля) до длины массива минус 1 (до текущего значения счетчика).

Утверждения о равенстве этих подмассивов преобразуются в леммы, которые поступают на вход системы доказательства. В результате образуется множество D пар подмассивов, равенство которых удалось доказать. Первым элементом каждой пары является подмассив массива A, а вторым элементом является подмассив массива B, для которого удалось доказать равенство первому элементу пары.

Каждый предикат из постусловия проверяется на выполнение свойства конкатенации или свойства конкатенации со склейкой на границе по предикату, найденному с помощью синтаксического анализа теории предметной области. Для каждого предиката S, удовлетворяющего любому из этих свойств, выполним следующие действия. Рассматриваемая стратегия генерирует леммы о выполнении свойства S для вторых элементов пар D, исходя из выполнения

свойства S для первых элементов пар D. Для предиката со свойством конкатенации со склейкой на границе стратегия генерирует леммы о выполнении условия склейки на границах вторых элементов пар D. Такие леммы помогают системе ACL2 доказать выполнение свойства S для подмассивов, полученных в результате объединения вторых элементов пар D. Следовательно, такие леммы заданы для автоматизации доказательства выполнения свойства конкатенации для результирующего массива.

2.3.3. Стратегия для программ с финитными итерациями над массивами

Пусть $a[i : j]$ — обозначение подмассива a от элемента с индексом i до элемента с индексом j (включая элементы $a[i]$ и $a[j]$). Пусть n — длина массива a. Пусть j-я итерация цикла является финитной итерацией над $a[0 : j-1]$ и $j+1$-я итерация является финитной итерацией над $a[0 : j]$. Такая ситуация может означать, что подмассив $a[j+1 : n-1]$ после исполнения j-й итерации равен подмассиву $a[j+1 : n-1]$ после исполнения $j+1$-й итерации. В этом случае будем называть подмассив $a[j+1 : n-1]$ необработанной частью массива $a[0 \ldots n-1]$. Аналогично будем называть подмассив $a[0 : j]$ обработанной частью массива $a[0 : n-1]$. Проверим истинность леммы о равенстве необработанной части массива после двух смежных итераций [121]. В случае успеха добавим эту лемму в теорию предметной области.

Стратегия генерирует лемму для каждой финитной итерации. Пусть такой цикл встречается i-м в коде программы. Сгенерируем лемму:

$$(P \land j \in N \land 0 < j < length(a)) \to$$
$$(rep_i(j, a, args).a)[j+1 : length(a)-1] = (rep_i(j+1, a, args).a)[j+1 : length(a)-1],$$

где

- P — предусловие.
- a — массив, над которым исполняется финитная итерация.

- j — счетчик цикла for, реализующего финитную итерацию.
- N — множество натуральных чисел.
- $length$ — функция, вычисляющая длину массива.
- $args$ — аргументы rep_i, вычисленные с обратного прослеживания.
- $rep_i(j, a, args).a$ — массив a после j-й итерации.
- $rep_i(j + 1, a, args).a$ — массив a после $j + 1$-й итерации.

Такая лемма может быть полезна если предикаты из спецификации программы удовлетворяют свойству конкатенации. Так как свойство конкатенации определено с помощью разбиения массива на части, то доказательство равенства необработанных частей массива может облегчить доказательство выполнения свойства конкатенации.

2.4. Основные стратегии

Основные стратегии отличаются тем, что целью их разработки было применение для автоматизации доказательства условий корректности [117, 123, 125, 144]. Основные стратегии могут использовать результаты применения вспомогательных стратегий.

2.4.1. Стратегия интерактивного доказательства

Стратегия интерактивного доказательства помогает доказать истинность формул вида $a \to b$. Для этого берутся формулы вида $c \to d$, где c является подформулой a определенного вида с точностью до переименования переменных. Затем доказательство формулы $a \to b$ сводится к доказательству формулы $(a \wedge d) \to b$.

При этом, возникают проблемы, которые в некоторых случаях алгоритм пытается разрешить. Одной из таких проблем является то, что переменные

формул $a \to b$ и $c \to d$, связанные кванторами всеобщности, имеют разные имена. Другой проблемой является проверка того, что c является подформулой a определенного вида.

Заменив вхождения всех связанных квантором всеобщности переменных формул a и c на один и тот же уникальный идентификатор, алгоритм пытается сопоставить формулы a и c. Затем пытается сопоставить связанные квантором всеобщности переменные формул a и c. Потом, воспользовавшись данным сопоставлением, переименовывает связанные квантором всеобщности переменные в формуле d.

Опишем данный алгоритм [144]. Рассмотрим метод доказательства формулы ϕ вида: $\forall x_1 x_2 ... x_{m-1} x_m \quad a(x_1 x_2 ... x_{m-1} x_m) \to b(x_1 x_2 ... x_{m-1} x_m)$, где

$$a = H_1(x_{K_{1_1}} x_{K_{1_2}} ... x_{K_{1_{l_1-1}}} x_{K_{1_{l_1}}}) \wedge H_2(x_{K_{2_1}} x_{K_{2_2}} ... x_{K_{2_{l_2-1}}} x_{K_{2_{l_2}}}) \wedge ...$$
$$\wedge H_{n-1}(x_{K_{n-1_1}} x_{K_{n-1_2}} ... x_{K_{n-1_{l_{n-1}-1}}} x_{K_{n-1_{l_{n-1}}}}) \wedge H_n(x_{K_{n_1}} x_{K_{n_2}} ... x_{K_{n_{l_n-1}}} x_{K_{n_{l_n}}}),$$

при этом $K_1 \cup K_2 \cup ... \cup K_{n-1} \cup K_n = \{1, ..., m\}$ и $|K_1| = l_1, |K_2| = l_2, ..., |K_{n-1}| = l_{n-1}, |K_n| = l_n$.

На входе алгоритма формула ϕ, набор аксиом и истинных формул:

$$\forall y_{11} y_{12} ... y_{1p_1-1} y_{1p_1} \ f_1, \quad \forall y_{21} y_{22} ... y_{2p_2-1} y_{2p_2} \ f_2 ...$$
$$\forall y_{q-11} y_{q-12} ... y_{q-1p_{q-1}-1} y_{q-1p_{q-1}} \ f_{q-1}, \quad \forall y_{q1} y_{q2} ... y_{qp_q-1} y_{qp_q} \ f_q$$

На выходе алгоритма: "формула ϕ истинна" или "неизвестно". Заметим, что случай ложности формулы ϕ относится к случаю "неизвестно".

1. Введем i:=1

2. Рассмотрим $\forall y_{i1} y_{i2} ... y_{ip_i-1} y_{1p_i} \ f_i$.

   Если f_i имеет вид $c(y_{i1} y_{i2} ... y_{ip_i-1} y_{1p_i}) \to d(y_{i1} y_{i2} ... y_{ip_i-1} y_{1p_i})$, то переход на шаг 3, иначе переход на шаг 9.

3. Если c имеет вид

$$G_1(y_{R_{1_1}} y_{R_{1_2}} ... y_{R_{1_{s_1-1}}} y_{R_{1_{s_1}}}) \wedge G_2(y_{R_{2_1}} y_{R_{2_2}} ... y_{R_{2_{s_2-1}}} y_{R_{2_{s_2}}}) \wedge ...$$
$$\wedge G_{t-1}(y_{R_{t-1_1}} y_{R_{t-1_2}} ... y_{R_{t-1_{s_{t-1}-1}}} y_{R_{t-1_{l_{t-1}}}}) \wedge G_t(y_{R_{t_1}} y_{R_{t_2}} ... y_{R_{t_{s_t-1}}} y_{R_{t_{l_t}}}),$$

где $t \leqslant n$, при этом $R_1 \cup R_2 \cup ... \cup R_{t-1} \cup R_t = \{i1, i2, ..., ip_i\}$ и $|R_1| = s_1, |R_2| = s_2, ..., |R_{t-1}| = s_{t-1}, |R_t| = s_t$, то переход на шаг 4, иначе переход на шаг 9.

4. Рассмотрим подформулу ϕ $a' = H'_1(x') \wedge H'_2(x') \wedge ... \wedge H'_{n-1}(x') \wedge H'_n(x')$, где каждый конъюнкт $H'_i(x')$ получен из конъюнкта $H_i(x_{K_{i_1}} x_{K_{i_2}} ... x_{K_{i_{l_i-1}}} x_{K_{i_{l_i}}})$ путем подстановки уникального идентификатора x' на место всех вхождений $x_{K_{i_1}} x_{K_{i_2}} ... x_{K_{i_{l_i-1}}} x_{K_{i_{l_i}}}$.

   Рассмотрим подформулу $c' = G'_1(x') \wedge G'_2(x') \wedge ... \wedge G'_{t-1}(x') \wedge G'_t(x')$, где каждый конъюнкт $G'_i(x')$ получен из конъюнкта $G_i(y_{R_{i_1}} y_{R_{i_2}} ... y_{R_{i_{s_i-1}}} y_{R_{i_{s_i}}})$ путем подстановки уникального идентификатора x' на место всех вхождений переменных $y_{R_{i_1}} y_{R_{i_2}} ... y_{R_{i_{s_i-1}}} y_{R_{i_{s_i}}}$.

   Рассмотрим множество биекций $e_1 e_2 ... e_{v-1} e_v$, где для любого i e_i — биекция между множеством индексов конъюнктов $\{1, .., t\}$ подформулы c и подмножеством U индексов конъюнктов подформулы a, при этом $e_i(w) = u \iff G'_w(x')$ синтаксически совпадает с $H'_u(x')$.

5. Введем $j := 1$

6. Сопоставляя $\forall v$ подформулы G_v и $H_{e_j(v)}$, построим таблицу соответствия между всеми переменными y формулы c и встречающимися в формуле $H_{e_j(1)} \wedge H_{e_j(2)} \wedge ... \wedge H_{e_j(t-1)} \wedge H_{e_j(t)}$ переменными x. Если не удалось построить такой корректной таблицы w, то переход на шаг 8, иначе переход на шаг 7.

7. Рассмотрим формулу

$$d'(w(y_{i1}) w(y_{i2}) ... w(y_{ip_i-1}) w(y_{1p_i})) = d[y_{i1} \leftarrow w(y_{i1}), y_{i2} \leftarrow w(y_{i2}), ..., y_{ip_i-1} \leftarrow w(y_{ip_i-1}), y_{1p_i} \leftarrow w(y_{1p_i})],$$

   полученную путем одновременной подстановки вместо $y_{i1} y_{i2} ... y_{ip_i-1} y_{1p_i}$ соответствующих им переменных $w(y_{i1}) w(y_{i2}) ... w(y_{ip_i-1}) w(y_{1p_i})$.

   Рассмотрим следующую формулу: $\forall x_1 x_2 ... x_{m-1} x_m \quad a \wedge d' \rightarrow b$

Истинность данной формулы может быть доказана по индукции с помощью отдельного доказательства базы индукции и отдельного доказательства шага индукции на SMT-решателе. Если в результате такой проверки получается "unsat", то переход на шаг 11, иначе переход на шаг 8.

8. Пусть $j := j + 1$. Если $j \leqslant v$, то переход на шаг 6, иначе переход на шаг 9.

9. Пусть $i := i + 1$. Если $i \leqslant q$, то переход на шаг 2, иначе переход на шаг 10.

10. Результатом работы алгоритма является "неизвестно".

11. Результатом работы алгоритма является "формула ϕ истинна".

Предложенный алгоритм позволяет расширить теорию новыми леммами. Данный алгоритм был реализован для управления доказательством УК, используя SMT-решатели Simplify, Z3 и CVC4. Но данный алгоритм возможно использовать и с другими системами доказательства.

2.4.2. Стратегия усиления условий корректности

Основная идея этой стратегии состоит в том, чтобы система доказательства попыталась доказать не исходное УК, а усиленное УК. Таким образом, стратегия состоит в применении с (возможными) повторениями преобразований, усиливающих формулу [4]. При этом должно выполняться следующее свойство: из истинности полученной формулы должна следовать истинность исходной формулы.

Главным преобразованием, усиливающим формулу, является замена констант и переменных в заключении импликаций на применения функций. Эта замена выполняется с использованием свойств транзитивности/нетранзитивности различных видов отношений сравнения. Если в качестве аргументов таких функций оказывается переменная, по которой ведется индукция, то это упрощает в

случае системы ACL2 применение индукционной гипотезы. Трудности у системы ACL2 вызывает сравнение значения применения функции с константой или переменной. В таком случае индукция по переменной, являющейся аргументом такого применения функции, может привести к тому, что система ACL2 не будет использовать индукционную гипотезу.

Опишем стратегию усиления УК [125], разработанную для того, чтобы помочь системам доказательства теорем автоматически доказывать условия корректности такого типа, которые возникают при применении символического метода верификации финитных итераций. Идея метода состоит в том, чтобы по условию корректности построить усиленное условие (т.е. более сильное логически, но не эквивалентное в общем случае) и доказать именно это усиленное условие: пусть ϕ — сгенерированное условие корректности (представляющее собой пропозициональную комбинацию равенств и неравенств термов); по ϕ строится более сильное условие θ, такое, что $\theta \to \phi$ — тождественно-истинное (по построению) утверждение; тогда, если система доказательства сможет доказать истинность условие θ, то исходное условие ϕ тоже истинно.

Сгенерированные условия корректности могут использовать функции как предметной области, так и специальные функции замены rep_i, представляющие в символической форме действие цикла [114]. Каждая rep_i соответствует своей программной переменной, изменяемой в цикле. Кроме того, особую роль при применении символического метода верификации финитных итераций может играть и переменная, соответствующая длине рассматриваемой части массива, так как именно по этой переменной ведется индукция в доказательстве. Кроме того, при построении функций rep_i используются начальные значения соответствующих программных переменных до исполнения цикла и может использоваться условие выхода из цикла.

Ниже описан алгоритм верификации условий корректности, основанный на поэтапном преобразовании условия корректности ϕ в усиленное условие корректности θ, причем на любом этапе могут выполняться неэквивалентные, но усиливающие преобразования: истинность построенного условия достаточна для истинности условия корректности. Те или иные преобразования выбраны

так, чтобы избежать случаев, вызывающих трудности для систем доказательства. Например, в качестве таких трансформаций используется явная подстановка тела нерекурсивной функции вместо ее применения.

Входными данными алгоритма являются формула ϕ, представляющая условие корректности, имя переменной, соответствующей длине рассматриваемой части массива (обозначим имя этой переменной символом "n"), определения функций rep_i (k обозначает их количество), и теория предметной области, а также начальные значения программных переменных до исполнения цикла и условие выхода из цикла.

Результатом работы алгоритма будет являться либо утверждение "формула ϕ истинна" в случае, если система доказательства смогла доказать усиленное условие корректности θ, либо "неизвестно" в случае, если система доказательства не смогла доказать усиленное условие корректности θ.

Опишем алгоритм. Он представляет собой последовательное выполнение следующих шести шагов:

1. Приведем формулу ϕ к форме конъюнкции клозов.

2. Для каждого клоза построим граф отношений между переменными и применениями функций в его посылке. Вершинами данного графа являются переменные и применения функций из посылки рассматриваемого клоза. Так как посылка клоза — это конъюнкция гипотез, то при построении ребер между вершинами происходит анализ этих гипотез. Вершины a и b данного графа соединяются ребром (a, b) с пометкой R из множества $\{=, \neq, <, >, \leq, \geq\}$, если a или b являются переменной, а в посылке клоза есть гипотеза $R(a, b)$. Таким образом, ребра представляют отношения между переменными и применениями функций, перечисленные в посылке клоза.

   Для каждого клоза, каждой переменной v и каждого отношения R из множества $\{=, \neq, <, >, \leq, \geq\}$ естественным образом определим специальную процедуру поиска (ближайшего или соответствующего) применения функции и списка гипотез, подтверждающих это применение. Аргументом

такой процедуры является граф G отношений рассматриваемого клоза, переменная v и отношение R. Возвращаемыми значениями являются либо сообщение "соответствующее применение функции не найдено", либо применение функции и специальный список гипотез из посылки клоза. В зависимости от отношения R работа процедуры сводится к следующему:

а. если R — это $=$, то пусть F — это множество вершин графа G, которые достижимы из v по отношению $(=)^*$ (т.е. вдоль ребер, помеченных отношением $=$);

б. если R — это $\leq$, то пусть F — это множество вершин графа G, которые достижимы из v по отношению $(= \cup < \cup \leq)^*$ (т.е. вдоль ребер, помеченных отношениями $=$, $<$ и $\leq$);

в. если R — это $\geq$, то пусть F — это множество вершин графа G, которые достижимы из v по отношению $(= \cup > \cup \geq)^*$ (т.е. вдоль ребер, помеченных отношениями $=$, $>$ и $\geq$);

г. если R — это $<$, то пусть F — это множество вершин графа G, которые достижимы из v по отношению $(= \cup < \cup \leq)^* \circ (<) \circ (= \cup < \cup \leq)^*$ (т.е. вдоль ребер, помеченных отношениями $=$, $<$ и $\leq$, среди которых обязательно есть одно, помеченное $<$);

д. если R — это $>$, то пусть F — это множество вершин графа G, которые достижимы из v по отношению $(= \cup > \cup \geq)^* \circ (>) \circ (= \cup > \cup \geq)^*$ (т.е. вдоль ребер, помеченных отношениями $=$, $>$ и $\geq$, среди которых обязательно есть одно, помеченное $>$);

е. если R — это $\neq$, то пусть F — это множество вершин графа G, которые достижимы из v по отношению $(=)^* \circ (\neq \cup < \cup >) \circ (=)^*$ (т.е. вдоль ребер, помеченных отношением $=$, среди которых обязательно есть ровно одно, помеченное либо как $\neq$, либо как $<$, либо как $>$).

Если среди этих вершин F есть хотя бы одно применение функции, то вернуть ближайшее (по числу использованных гипотез) к v такое применение

и список всех использованных равенств вдоль этого пути; в противном случае результатом является сообщение "соответствующее применение функции не найдено".

После построения графа отношений для клоза происходит обработка заключения этого клоза. Поскольку заключение представляет собой дизъюнкцию целей, то при обработке заключения происходит анализ целей клоза, каждая из которых имеет вид $R(c, d)$, где R — отношения из множества $\{=, \neq, <, >, \leq, \geq\}$, при этом c и d — константы, переменные или применения функции. Пусть $R(c, d)$ — одна из целей. Введем вспомогательные переменные v и w и инициализируем их значениями c и d, соответственно, а также вспомогательные переменные q и r и инициализируем их обе пустыми списками. Далее, если первый аргумент c отношения R является переменной, то тогда запускается специальная процедура поиска соответствующего переменной применения функции для этой переменной c и отношения R; в случае успеха соответствующее применение функции и специальный список конъюнктов присваиваются переменным v и q, соответственно. Потом рассмотрим второй аргумент d отношения R. Если v совпадает с c или g — это не $\neq$, а d — это переменная, то тогда запускается специальная процедура поиска соответствующего переменной применения функции: если отношение R — это $=$ или $\neq$, то в качестве аргументов этой процедуре передаются d и R; если отношение R — это $<$ или $\leq$, то в качестве аргументов процедуре передаются d и "противоположные" отношения $>$ или $\geq$, соответственно; если отношение R — это $>$ или $\geq$, то в качестве аргументов процедуре передаются d и "противоположные" отношения $<$ или $\leq$, соответственно; в случае успеха соответствующее применение функции и специальный список конъюнктов присваиваются переменным w и r, соответственно. Если начальное значение переменной v или переменной w было изменено, то цель $R(c, d)$ заменяется на цель $(= \quad v \quad w)$. Если при этом отношение R — это $=$, то из посылки рассматриваемого клоза удаляются все гипотезы, встречающиеся в хотя бы

в одном из списков q или r.

3. Для тех rep_i, для которых удалось установить, что их можно переопределить нерекурсивным способом, выполним явную подстановку. Для этого достаточно доказать, что из невыполнения условия выхода из цикла следует равенство rep_i и начального значения соответствующей программной переменной. Для каждой такой функции построим дерево, представляющее ее тело. Внутренними вершинами такого дерева являются условные операторы, а листьями – возвращаемые значения функции; левым потомком такого оператора является его значение при выполнении его условия, а ребро между этими вершинами помечается условием данного оператора; правым потомком такого оператора является его значение при невыполнении его условия, а ребро между этими вершинами помечается отрицанием условия данного оператора.

4. Для каждого клоза происходит обработка его заключения. Так как такое заключение представляет собой дизъюнкцию целей, то при обработке заключения происходит анализ каждой индивидуальной цели. Пусть g — одна из этих целей, а c — является применением нерекурсивной функции, входящим в g. Тогда происходит процедура замены цели g на конъюнкцию специальных импликаций. Рассмотрим множество промежуточных импликаций. Каждая промежуточная импликация взаимно соответствует листу в представляющем тело рассматриваемой функции дереве. Ее посылкой является конъюнкция всех пометок ребер на пути от вершины дерева до данного листа. Ее заключением является цель g, где каждое вхождение c заменено на представленное листом возвращаемое значение рассматриваемой функции. Для каждой промежуточной импликации процедура замены строит специальную импликацию путем подстановки аргументов применения функции c вместо переменных тела данной функции. После каждой подстановки каждое заключение приводится к форме клоза. Данный шаг повторяется до тех пор, пока в заключениях клозов присутствуют применения нерекурсивных функций.

5. Если на предыдущих шагах (2—4) формула изменилась, то повторить эти шаги еще раз.

6. Доказательство полученной формулы с помощью системы доказательства. Если формулу удалось доказать, то результатом работы алгоритма является утверждение "формула ϕ истинна", иначе "неизвестно".

Данная стратегия была реализована в C-lightVer для системы ACL2. Подчеркнем, что данный алгоритм можно обобщить для его использования не только с системой ACL2, но и с другими системами проверки формул на истинность (в качестве примера можно рассмотреть SMT-решатели CVC4 и Z3).

2.4.3. Теорема о корректности стратегии усиления условий корректности и доказательство данной теоремы

Отметим, что алгоритм всегда завершает свою работу [4]. Поэтому, результатом применения стратегии всегда является либо сообщение "формула ϕ доказана" или сообщение "неизвестно". Доказательство заключается в том, что на каждом цикле стратегии (последовательное исполнение шагов (2)–(4)) строго убывает количество вхождений в формулу применений тех функций, которые можно переопределить нерекурсивным образом. Количество таких вхождений не может быть меньше нуля.

Так как стратегия применяет преобразования, в результате которых полученная формула не эквивалентна исходной, то было необходимо доказать, что полученная формула является усилением исходной. Поэтому, была сформулирована и доказана теорема о корректности стратегии усиления УК [123].

Теорема 1. *Стратегия усиления УК корректна.*

Приведем доказательство данной теоремы. Стратегия применяется к условию корректности ϕ.

Доказательство. Пусть в результате исполнения стратегии доказательства получена формула ψ. Тогда корректность стратегии эквивалентна истинности

формулы $\psi \to \phi$. Введем понятие хода стратегии. Ход стратегии – это исполнение шага стратегии. Пронумеруем ходы стратегий от 1 до l. Так как шаги стратегии могут исполняться повторно, то номер хода может не совпадать с номером шага стратегии. Пусть нулевой ход стратегии соответствует состоянию до начала исполнения стратегии. Пусть ψ_0 – преобразуемая формула до исполнения стратегии. По определению стратегии формулы ψ_0 и ϕ совпадают. Пусть ψ_i – результат i-го хода стратегии, где $1 \leq i \leq l$. Поэтому формулы ψ_l и ψ совпадают. Для доказательства корректности стратегии докажем формулу $\forall i (0 \leq i \leq l) \psi_i \to \phi$ индукцией по i.

Так как $\phi = \psi_0$, то база индукции верна.

Пусть формула $\psi_{i-1} \to \phi$, где $i > 0$, истинна. После i-го хода стратегии была получена формула ψ_i. Докажем, что формула $\psi_i \to \phi$ истинна. Для этого рассмотрим варианты соответствия между i-м ходом стратегии и шагом стратегии:

- Пусть на i-м ходе стратегии был исполнен шаг 1. Так как преобразования шага 1 сохраняют эквивалентность, то $\psi_i \iff \psi_{i-1}$. Так как формула $\psi_{i-1} \to \phi$ истинна, то формула $\psi_i \to \phi$ истинна.

- Пусть на i-м ходе стратегии был исполнен шаг 3. Так как преобразования шага 3 сохраняют эквивалентность, то $\psi_i \iff \psi_{i-1}$. Так как формула $\psi_{i-1} \to \phi$ истинна, то формула $\psi_i \to \phi$ истинна.

- Пусть на i-м ходе стратегии был исполнен шаг 4. Так как преобразования шага 4 сохраняют эквивалентность, то $\psi_i \iff \psi_{i-1}$. Так как формула $\psi_{i-1} \to \phi$ истинна, то формула $\psi_i \to \phi$ истинна.

- Пусть на i-м ходе стратегии был исполнен шаг 2. Преобразования шага 2 могут не сохранять эквивалентность. Далее преобразованием будем называть преобразования, выполняемые на шаге 2. Отметим, что по определению стратегии на $i-1$-м ходе был исполнен шаг 1, поэтому формула ψ_{i-1} имеет вид $X_1 \wedge ... \wedge X_k$.

Так как каждый конъюнкт преобразуется отдельно, то формула ψ_i имеет вид $X'_1 \wedge ... \wedge X'_k$, где $\forall j(1 \leq j \leq k)$ конъюнкт X'_j является результатом применения стратегии 2 к конъюнкту X_j. Отметим, что X'_j может совпадать с X_j. Для доказательства формулы $\psi_i \to \phi$ достаточно доказать $\psi_i \to \psi_{i-1}$. Поэтому для любого конъюнкта X_j докажем, что $X'_j \to X_j$.

Так как на $i-1$-м ходе стратегии был исполнен шаг 1, то конъюнкт X_j имеет вид $Y_1 \wedge ... \wedge Y_m \to Z_1 \vee ... \vee Z_n$. Пусть $\forall k(1 \leq k \leq m)$ формула Y_i называется гипотезой. Пусть $\forall k(1 \leq k \leq n)$ формула Z_k называется целью.

Согласно определению шага 2, строится граф соотношений G. Так как каждая цель преобразуется отдельно, то конъюнкт X'_j имеет вид $Y_{U_1} \wedge ... \wedge Y_{U_s} \to Z'_1 \vee ... \vee Z'_n$, где $\forall k(1 \leq k \leq n)$ цель Z'_k является результатом применения стратегии 2 к цели Z_k. Отметим, что Z'_k может совпадать с Z_k. Множество U является подмножеством $\{1, ..., m\}$, s – мощность множества U. Множество U введено из-за возможного удаления гипотез на этапе преобразования. Пусть множество V – разница множеств $1, ..., m$ и U. Обозначим мощность множества V как t. Отметим, что $t = m - s$. Для доказательства формулы $X'_j \to X_j$ достаточно для любой цели Z_k доказать, что

$$(Y_{U_1} \wedge ... \wedge Y_{U_s} \to Z'_k) \to (Y_1 \wedge ... \wedge Y_m \to Z_k)$$

Так как импликация правоассоциативна, то эта формула эквивалентна следующей:

$$((Y_{U_1} \wedge ... \wedge Y_{U_s} \to Z'_k) \wedge Y_1 \wedge ... \wedge Y_m) \to Z_k$$

Получим эквивалентную формулу, переупорядочив конъюнкты:

$$((Y_{U_1} \wedge ... \wedge Y_{U_s} \to Z'_k) \wedge Y_{U_1} \wedge ... \wedge Y_{U_s}) \wedge Y_{V_1} \wedge ... \wedge Y_{V_t} \to Z_k$$

По закону дистрибутивности эта формула эквивалентна следующей:

$$((Y_{U_1} \wedge ... \wedge Y_{U_s} \wedge Z'_k) \wedge Y_{V_1} \wedge ... \wedge Y_{V_t} \to Z_k$$

Получим эквивалентную формулу, переупорядочив конъюнкты:

$$(Y_1 \wedge ... \wedge Y_m \wedge Z'_k) \to Z_k$$

Докажем данную формулу. Отметим, что корректность стратегии не зависит от множества V. Удаление гипотез является эвристикой, ориентированной на помощь системе доказательства.

Рассмотрим два случая:

1. Цель Z_k не имеет вид $R(c, d)$, где R — отношения из множества $\{=, \neq, <, >, \leq, \geq\}$, при этом c и d — константы, переменные или применения функции. Тогда преобразование не изменяет цель Z_k. Поэтому формулы Z'_k и Z_k совпадают. Следовательно, формула $(Y_1 \wedge ... \wedge Y_m \wedge Z'_k) \to Z_k$ истинна.

2. Цель Z_k имеет вид $R(c, d)$, где R — отношения из множества $\{=, \neq, <, >, \leq, \geq\}$, при этом c и d — константы, переменные или применения функции. В этом случае необходимо рассмотреть несколько подслучаев. Для простоты и краткости изложения рассмотрим только один подслучай, остальные подслучаи рассматриваются аналогично.

   Рассмотрим подслучай, когда R — отношение $\neq$, c — переменная, d – применение функции. Таким образом, цель Z_k имеет вид $c \neq d$. В рассматриваемом подслучае доказываемая формула эквивалентна следующей:

   $$Y_1 \wedge ... \wedge Y_m \wedge Z'_k \to c \neq d$$

   Согласно определению шага 2, введем переменные v и w. Их начальные значения – c и d соответственно. Кроме того, согласно определению шага 2, введем переменные q и r. Пустой список является начальным значением этих переменных.

Так как c – переменная, то исполняется специальная процедура поиска с аргументами G, c и $\neq$. Рассмотрим возможные результаты работы такой процедуры.

– Возвращаемым значением является сообщение "соответствующее применение функции не найдено". Поэтому значения переменных v, w, q и r не изменяются. Так как d – применение функции, то специальная процедура поиска для цели $c \neq d$ исполняется только один раз. В итоге значениями переменных v, w являются c и d, соответственно, а значениями q и r являются пустые списки. Значит, формулы Z'_k и Z_k совпадают. Поэтому формула $Y_1 \wedge ... \wedge Y_m \wedge Z'_k \rightarrow c \neq d$ имеет вид $Y_1 \wedge ... \wedge Y_m \wedge c \neq d \rightarrow c \neq d$. Эта формула истинна.

– Возвращаемыми значениями являются выражение h и список W, где h – применение функции, а W – список гипотез. Пусть A – список индексов гипотез в порядке их расположения в списке w. Длины списков w и A совпадают; обозначим их как p. Пусть множество B содержит все индексы из множества $1, ..., m$, не входящие в список A. Мощность множества B обозначим как f. Отметим, что $m = f + p$. Тогда формула $Y_1 \wedge ... \wedge Y_m \wedge Z'_k \rightarrow c \neq d$ эквивалентна следующей

$$Y_{B_1} \wedge ... \wedge Y_{B_f} \wedge Y_{A_1} ... \wedge Y_{A_p} \wedge Z'_k \rightarrow c \neq d$$

Так как d – применение функции, то специальная процедура поиска для цели $c \neq d$ исполняется только один раз. В итоге значением переменной v является h, значением переменной w является d, значением переменной q является W, а значением переменной r – пустой список. Поэтому формула Z'_k имеет вид $h = d$. Значит, получим следующую эквивалентную формулу:

$$Y_{B_1} \wedge ... \wedge Y_{B_f} \wedge Y_{A_1} ... \wedge Y_{A_p} \wedge h = d \rightarrow c \neq d$$

Докажем, что существует индекс e, такой что $1 \leq e \leq p$ и Y_{A_e} имеет вид $O(f_e, f_{e+1})$, где отношение O – один из видов неравен-

ства ($\neq$ или $<$ или $>$), $f_e = c$ и $f_{e+1} = h$. Предположим, что индекса e с такими свойствами не существует. Тогда специальная процедура поиска с аргументами G, c и $\neq$ вернет сообщение "соответствующее применение функции не найдено". Но это противоречит тому, что возвращаемыми значениями являются выражение h и список W. Значит, рассматриваемое предположение неверно, и индекс e с такими свойствами существует. Поэтому рассматриваемая формула имеет вид:

$$Y_{B_1} \wedge ... \wedge Y_{B_f} \wedge Y_{A_1} ... \wedge O(f_e, f_{e+1}) \wedge ... \wedge Y_{A_p} \wedge h = d \rightarrow c \neq d$$

Для доказательства данной формулы достаточно доказать, что

$$Y_{B_1} \wedge ... \wedge Y_{B_f} \wedge Y_{A_1} ... \wedge c \neq h \wedge ... \wedge Y_{A_p} \wedge h = d \rightarrow c \neq d$$

Данная формула истинна по свойствам равенств и неравенств. Так как данная формула истинна, то формула $Y_1 \wedge ... \wedge Y_m \wedge Z'_k \rightarrow c \neq d$ истинна.

Значит, формула $Y_1 \wedge ... \wedge Y_m \wedge Z'_k \rightarrow c \neq d$ истинна. Так как из истинности формулы $(Y_1 \wedge ... \wedge Y_m \wedge Z'_k) \rightarrow Z_k$ следует истинность формулы $X'_j \rightarrow X_j$, то стратегия корректна, что и требовалось доказать.

□

2.4.4. Стратегия для программ, постусловием которых является разбор случаев выхода из цикла

Идея данной стратегии состоит в проверке предположения, что постусловие программы описывает случаи в форме конъюнкции импликаций, случился или нет выход из тела цикла [117]. Система доказательства проверяет истинность такого предположения. Если оно истинно, то это помогает более точно описать случаи в постусловии. Доказательство всех уточненных случаев помогает доказать условие корректности.

На вход алгоритма поступает условие корректности ϕ, переменная n, определение функции rep, теория предметной области и постусловие.

В результате работы алгоритм выдает либо «ϕ истина» или «неизвестно». Данный алгоритм выглядит следующим образом:

1. Пусть M обозначает кортеж импликаций, являющихся конъюнктами постусловия. Рассмотрим кортеж N, такой что i-й элемент N является посылкой i-го элемента M. Переход на шаг 2.

2. Для каждого элемента из N выполнить шаг 3. Если результат истинен, добавить в теорию лемму, представляющую собой конъюнкцию ϕ и равенства i-го элемента N с $rep(\ldots).loop\text{-}break$. Иначе перейти на шаг 4. Если результат истинен, добавить в теорию лемму, представляющую собой конъюнкцию ϕ и равенства i-го элемента N с $\neg rep(\ldots).loop\text{-}break$. Перейти на шаг 5.

3. Пусть θ обозначает i-й элемент N. Пусть ω является конъюнкцией ϕ и равенства θ с $rep(\ldots).loop\text{-}break$. Проверить истинность ω в ACL2. Если ω была доказана, то вернуть «истина», иначе — «ложь».

4. Пусть θ обозначает i-й элемент N. Пусть ω является конъюнкцией ϕ и равенства θ с $\neg rep(\ldots).loop\text{-}break$. Проверить истинность ω в ACL2. Если ω была доказана, то вернуть «истина», иначе — «ложь».

5. Проверить истинность ϕ в ACL2 с помощью полученных лемм. Если ϕ была доказана, то вернуть «ϕ истина», иначе — «неизвестно».

Вместо ACL2 в данной стратегии могут быть использованы другие системы доказательства, язык которых позволяет определять функцию rep, возвращающую информацию, исполнилась ли инструкция `break`.

2.5. Стратегии для классов финитных итераций

От рассмотрения более общих стратегий перейдем к рассмотрению стратегий для конкретных классов финитных итераций [118, 121, 126]. Данные стратегии также могут использовать результаты применения вспомогательных стратегий.

2.5.1. Стратегия для финитных итераций с инструкцией `break`

Введем обозначения. Стратегия принимает в качестве аргумента финитную итерацию над массивом a длины n. Таким образом, в условии корректности вызов функции *rep* имеет следующие аргументы:

- Аргумент *iteration* соответствует номеру итерации.
- Аргумент *env* соответствует вектору неизменяемых параметров цикла.
- Аргумент fr соответствует вектору v.

Условием применимости стратегии для финитных итераций с инструкцией `break` [118] является наличие финитной итерации и наличием в ее теле инструкции `break`. Назовем `break`-условием конъюнкцию условий (выполним подстановки при наличии присваиваний) на пути к инструкции `break`. Фактически `break`-условие является функцией $\mathit{br\text{-}cond}(iteration, env, fr)$.

Значение поля *loop-break* структуры $frame$ для определенной итерации является индикатором, сработал ли оператор `break` ранее. Данная стратегия состоит в попытке доказать набор вспомогательных утверждений о срабатывании оператора `break`:

1. Если $rep(iteration, env, fr).\mathit{loop\text{-}break}$, тогда $rep(iteration, env, fr).i = rep(\ iteration - 1, env, fr).i$.

2. Если $\neg rep(iteration, env, fr).\mathit{loop\text{-}break}$, тогда $rep(iteration, env, fr).i = env.\ \mathit{upper\text{-}bound} - iteration - 1$.

3. Если $rep(iter, env, fr).\mathit{loop\text{-}break}$ и $iter \leq iteration$, тогда $rep(iteration, env, fr).\mathit{loop\text{-}break}$.

4. Если $rep(iter, env, fr).\mathit{loop\text{-}break}$ и $iter \leq iteration$, тогда $rep(iter, env, fr) = rep(iteration, env, fr)$.

5. Если $\neg rep(iteration, env, fr).loop\text{-}break$ и $iter \leq iteration$, тогда $\neg rep(iter, env, fr).loop\text{-}break$.

6. $\neg(br\text{-}iter - 1, fr, env).loop\text{-}break$ и $rep(br\text{-}iter - 1, fr, env).loop\text{-}break$.

7. Если $iter \in [br\text{-}iter : env.upper\text{-}bound]$, тогда $rep(env, iter, fr).loop\text{-}break$.

8. Если $iteration \in [0 : br\text{-}iter - 1]$, тогда $\neg rep(iteration, env, fr).loop\text{-}break$.

9. Если $iteration \in [0 : br\text{-}iter - 1]$, тогда $\neg br\text{-}cond(iteration, env, fr)$.

10. $iteration \in [br\text{-}iter : env.upper\text{-}bound] \Rightarrow br\text{-}cond(iteration, env, fr)$.

11. $\neg br\text{-}cond(br\text{-}iter - 1, env, fr)$ и $br\text{-}cond(br\text{-}iter, env, fr)$.

Данные утверждения обрабатываются с помощью стратегии обобщения посылки лемм о свойствах функции замены. Те из них, которые удается доказать, являются леммами и добавляются в предметную область. Они могут помочь системе ACL2 доказать условие корректности.

2.5.2. Стратегия для программ с выходом из цикла

Входными аргументами данной стратегии являются импликация ψ, содержащая применение функции $rep(n, \ldots)$, и ее посылка ϕ. Попытаемся доказать формулу

$$\phi \to rep(n, \ldots).loop\text{-}break \qquad (\psi\text{-}lemma\text{-}1)$$

индукцией по n. Если система ACL2 доказывает (ψ-*lemma-1*), то добавляем ее в теорию предметной области. Эта лемма означает, что посылка ϕ импликации ψ описывает один из случаев, когда при исполнении цикла происходит исполнение *break*. Попытаемся доказать ψ, используя (ψ-*lemma-1*) и индукцию по n.

Если система ACL2 не доказала (ψ-*lemma-1*), то попытаемся доказать формулу

$$\phi \to \neg rep(n, \ldots).loop\text{-}break \qquad (\psi\text{-}lemma\text{-}2)$$

индукцией по n. Если система ACL2 доказывает (ψ-*lemma-2*), то добавляем ее в теорию предметной области. Эта лемма означает, что посылка ϕ импликации

ψ описывает один из случаев, когда при исполнении цикла не происходит исполнение *break*. Попытаемся доказать ψ, используя (ψ-*lemma-2*) и индукцию по n.

Вместо ACL2 в данной стратегии могут быть использованы другие системы доказательства, язык которых позволяет определять функцию *rep*, возвращающую информацию, исполнилась ли инструкция `break`.

Отметим, что стратегия для программ с выходом из цикла похожа на стратегию для программ, постусловием которых является разбор случаев выхода из цикла [117]. Во-первых, обе стратегии основаны на использовании значения поля *loop-break*. Во-вторых, обе стратегии полностью автоматизированы, так как при применении данных стратегий от пользователя системы верификации не требуется предоставлять дополнительные данные. В-третьих, обе стратегии являются эвристическими методами доказательства, т.е. применение данных стратегий к истинной формуле может не привести к успешному доказательству.

Поэтому опишем различия между стратегиями. Во-первых, стратегия для программ с выходом из цикла принимает в качестве аргумента любую импликацию, содержащую *rep*. Стратегия для программ, постусловием которых является разбор случаев выхода из цикла, анализирует только постусловие программы. Во-вторых, стратегию для программ, постусловием которых является разбор случаев выхода из цикла, генерирует лемму в виде конъюнкции. Первый элемент такой конъюнкции – это УК, второй – равенство посылки импликации из постусловия и значения поля *loop-break*. Такая структура генерируемой леммы обусловлена тем, что такая стратегия применяется только тогда, когда постусловие программы представляет собой конъюнкцию импликаций, т.е. представляет собой разбор случаев, где каждый случай описывается посылкой импликации. Поэтому такая стратегия генерирует лемму более сложной структуры, чем стратегия для программ с выходом из цикла.

2.5.3. Стратегия для финитных итераций над изменяемыми массивами

Введем обозначения. Стратегия принимает в качестве аргумента финитную итерацию над массивом a длины n. Таким образом, в условии корректности вызов функции rep имеет следующие аргументы:

- Аргумент $iteration$ соответствует номеру итерации.
- Аргумент env соответствует вектору неизменяемых параметров цикла.
- Аргумент fr соответствует вектору v.

Условием применимости данной стратегии является наличие финитной итерации и наличием в ее теле инструкций вида `a[expr-index] = expr-value`. Пусть эта итерация содержит w подобных присваиваний. С помощью функции $c2acl2$ переведем каждое выражение $expr\text{-}index_i$ в выражение $expr-ind_i$ языка ACL2 для каждого i $(1 \leq i \leq w)$.

Сгенерируем следующую формулу:

$$\begin{gathered}(iteration \in N) \wedge (index \in N) \wedge \\ (iteration \leq env.upper\text{-}bound) \wedge (env.upper\text{-}bound < (lenfr.a)) \wedge \\ (fr.j = (env.upper\text{-}bound - 1)) \\ (index \neq expr\text{-}ind_1) \\ \ldots \\ (index \neq expr\text{-}ind_w) \\ \rightarrow \\ rep.a(iteration - 1, env, fr)[index] = \\ rep.a(iteration, env, fr)[index]\end{gathered},$$

Если такую формулу удается доказать, то соответствующая лемма добавляется в предметную область. Данная лемма является утверждением, что элемент массива, индекс которого не совпадает с индексами левых частей присваивания, не изменяется при переходе на следующую итерацию. Такая лемма может быть полезна, если доказательство равенства подмассивов основано на индукции по параметру $iteration$.

2.5.4. Стратегия для программ с вложенными циклами

Рассмотрим случай, когда возрастание количества итераций внешнего цикла приводит к возрастанию части последовательности, обрабатываемой внутренним циклом. Тогда попытаемся доказать УК, используя индукцию по количеству итераций внешнего цикла.

Пусть УК имеет вид $P \to Q$, где

- Q — заключение УК, зависящее от rep_i.
- P — посылка УК.
- a — массив, над которым исполняется финитная итерация.
- n — длина массива.
- $args$ — аргументы rep_i, вычисленные с помощью обратного прослеживания.
- rep_i — функция замены для i-го цикла, тело которой содержит применение rep_{i+1}.

Стратегия [121] заключается в доказательстве такого УК индукцией по n. Такая стратегия может быть полезна, если внутренний цикл определяет ту часть последовательности, над которой исполняется каждая итерация внешнего цикла. Тогда подстановка вместо операции замены для внешнего цикла ее определения приводит к формуле, где доказываемое свойство сформулировано относительно внутреннего цикла. Доказывать это свойство позволяет индукционная гипотеза о части последовательности, обрабатываемой внутренним циклом.

2.6. Выводы

На основе сделанного описания алгоритмов генерации операций замены и стратегий доказательства УК можно сделать следующие выводы:

- Алгоритмы генерации функций, выражающих результат финитных итераций, создают операции замены rep, возвращающие дополнительную информацию о данных итерациях. В качестве примера можно привести информацию о том, исполнилась ли инструкция `break` или информацию о значении счетчика цикла при окончании исполнения итерации. Такая информация позволяет стратегиям доказательства УК генерировать леммы о структуре финитных итераций и свойствах операций замены rep.

- Все предложенные стратегии, кроме стратегии интерактивного доказательства, исполняются в автоматическом режиме. В процессе исполнения стратегии интерактивного доказательства пользователю необходимо предложить лемму, соответствующую определенному шаблону.

- Все предложенные стратегии, кроме стратегии усиления УК, основаны на генерации лемм о свойствах программ и финитных итераций, которые могут помочь доказать УК. Если истинность этих лемм удается доказать, то леммы добавляются в теорию предметной области и их можно использовать для доказательства целевой теоремы. Поэтому, для алгоритмов генерации лемм, на которых основаны такие стратегии, не требуется доказательство корректности. Стратегия усиления УК основана на генерации такой формулы, из истинности которой следует истинность УК. Поэтому, для такой стратегии была сформулирована и доказана теорема о корректности, которая утверждает, что сгенерированная формула является усилением УК.

Рассмотрим далее метод автоматизации локализации ошибок.

Глава 3

Метод автоматизации локализации ошибок

Опишем метод автоматизации локализации ошибок [19, 114, 121, 126], реализованный в системе C-lightVer. Данный метод включает в себя стратегии локализации ошибок [121, 126] и метод генерации текста о сопоставлении подформул условий корректности и фрагментов программы [19, 114].

3.1. Стратегии локализации ошибок

Для автоматизации локализации ошибок в программах с финитными итерациями комплексный подход был расширен стратегией проверки ложности недоказанных условий корректности [126], стратегией поиска циклов с неиспользуемыми присваиваниями элементам массива [121] и стратегией проверки исполнения инструкции `break` на первой итерации цикла [121]. Эти стратегии используются для проверки выполнения свойств циклов, которые могут означать наличие ошибок. Комплексный подход, реализованный в системе C-lightVer, позволяет генерировать объяснения для таких случаев.

3.1.1. Стратегия проверки ложности недоказанных условий корректности

Отметим, что доказательство ложности УК гарантирует наличие ошибки в программе или в ее спецификациях. В таком случае модуль локализации ошибок может сообщать о несоответствии программы и спецификации. Поэтому для более точного анализа УК необходимо разработать стратегии проверки их ложности. Так как переменные ACL2 находятся под неявным квантором всеобщности, то доказательство отрицания УК приводит к появлению квантора существования. Поэтому, другие стратегии комплексного подхода [117, 118, 123, 125] не подходят для доказательства отрицания УК. Проблема состоит также в том,

чтобы стратегия проверки ложности работала в случае цикла с оператором `break`.

Опишем стратегию проверки ложности недоказанных УК, разработанную для решения данных проблем [126].

Входным аргументом данной стратегии является УК (формула ω), содержащая применение функции $rep(n, \ldots)$, и имеющая следующую структуру:

$$\forall x_1 \ldots x_n \ (\phi_1(x_1 \ldots x_n) \rightarrow \psi_1(x_1 \ldots x_n)) \wedge \ldots \wedge (\phi_m(x_1 \ldots x_n) \rightarrow \psi_m(x_1 \ldots x_n))$$

Все переменные формулы находятся под квантором всеобщности из-за ориентированности данной стратегии на систему ACL2. Отметим, что в определении любой из формул ϕ_i или ψ_i могут использоваться не все переменные из набора $x_1 \ldots x_n$. Но обозначим в качестве параметров весь набор переменных, так как в определение любой формулы можно ввести использование любой новой переменной, записав формулу в конъюнкции с дизъюнкцией этой переменной и ее отрицания. Такое преобразование возможно, так как весь набор переменных находится под квантором всеобщности. Поэтому, без ограничения общности можно обозначать зависимость от всего набора переменных. Известно, что формула ложна тогда и только тогда, когда ее отрицание истинно. Поэтому докажем формулу $\neg\omega$:

$$(\exists x_1 \ldots x_n \ (\phi_1(x_1 \ldots x_n) \wedge \neg\psi_1(x_1 \ldots x_n))) \vee \ldots \vee (\exists x_1 \ldots x_n \ (\phi_m(x_1 \ldots x_n) \wedge \neg\psi_m(x_1 \ldots x_n)))$$

Для каждого $i (1 \leq i \leq m)$ подформулу

$$\exists x_1 \ldots x_n \ (\phi_i(x_1 \ldots x_n) \wedge \neg\psi_i(x_1 \ldots x_n))$$

обозначим как T_i. Для доказательства формулы $\neg\omega$ достаточно доказать любую формулу $T_i (1 \leq i \leq m)$.

Поэтому данная стратегия проверки ложности УК состоит в том, чтобы применять специальную процедуру доказательства к формулам T_i. Если удалось доказать любую формулу T_i, то результатом стратегии будет сообщение о ложности УК `Verification condition is false`. Если не удалось доказать

ни одну формулу T_i, то результатом стратегии будет сообщение о неизвестном результате `Unknown`.

Опишем специальную процедуру доказательства. Рассмотрим применение данной процедуры к формуле $T_i(1 \leq i \leq m)$. Для доказательства формулы

$$\exists x_1 \dots x_n \ (\phi_i(x_1 \dots x_n) \wedge \neg\psi_i(x_1 \dots x_n))$$

достаточно доказать формулу

$$(\exists x_1 \dots x_n \ \phi_i(x_1 \dots x_n)) \wedge (\forall x_1 \dots x_n(\phi_i(x_1 \dots x_n) \rightarrow \neg\psi_i(x_1 \dots x_n)))$$

которую обозначим как T_i'. Обозначим как U_i формулу $\exists x_1 \dots x_n \ \phi_i(x_1 \dots x_n)$. Обозначим как V_i $\forall x_1 \dots x_n(\phi_i(x_1 \dots x_n) \rightarrow \neg\psi_i(x_1 \dots x_n))$. Специальная процедура доказательства сводится к проверке истинности U_i и к доказательству V_i.

Отметим, что истинность формулы U_i проверяется пользователем. Таким образом, данная стратегия проверки ложности УК является не автоматической, а автоматизированной. Процесс доказательства U_i не был автоматизирован по следующим причинам. Во-первых, генерация УК в системе C-lightVer основана на вычислении слабейшего предусловия [117]. Поэтому формула U_i не может содержать операцию замены. Значит, формула U_i может содержать только известные пользователю функции из спецификации. Во-вторых, многие известные системы доказательства имеют проблемы при автоматическом доказательстве формулы с квантором существования. В-третьих, эвристика, реализованная в данной стратегии, основана на гипотезе о простой посылке и сложном заключении импликаций, сгенерированных в результате вычисления слабейшего предусловия. Исходя из данной гипотезы, в системе C-lightVer была автоматизирована более сложная задача доказательства V_i. Если пользователь не сообщил об истинности U_i, то будем считать, что формулу T_i не удалось доказать.

Если пользователь сообщил об истинности U_i, то попытаемся доказать V_i. Сначала рассмотрим случай, когда формула V_i не содержит применения функции rep. Так как система ACL2 ориентирована на доказательство формул, где

все переменные находятся под квантором всеобщности, то попытаемся автоматически доказать формулу V_i в системе ACL2. Теперь рассмотрим случай, когда формула V_i содержит применение функции $rep(n, \ldots)$. Так как пользователь не знает кода функции rep, то в этом случае возможно только автоматическое доказательство. Также проблемой является возможный выход из цикла, усложняющий код функции rep. Поэтому, попытаемся доказать V_i, используя стратегию для программ с выходом из цикла 2.5.2.

3.1.2. Стратегия поиска циклов с неиспользуемыми присваиваниями элементам массива

Пусть в цикле, реализующем финитную итерацию над массивом, содержатся присваивания элементам этого массива и значения элементов массива после исполнения цикла равны значениям элементов массива до исполнения цикла. Значит, эти присваивания могут быть неиспользуемыми операциями. Такая ситуация может означать наличие ошибки. Поэтому, можно предупредить пользователей системы верификации о таких присваиваниях и о содержащем такие присваивания цикле.

Стратегия поиска таких циклов проверяет каждый цикл над массивом с присваиваниями элементам массива [121]. Пусть такой цикл встречается i-м в коде программы. Стратегия основана на генерации и проверке истинности леммы $P \to (a = rep_i(a, args).a)$, где

- P — предусловие.
- a — массив, над которым исполняется финитная итерация.
- rep_i — операция замены для финитной итерации.
- $args$ — аргументы rep_i, вычисленные с помощью обратного прослеживания.
- $rep_i(a, args).a$ — массив a после исполнения цикла.

Если такую лемму удалось доказать, то с помощью метода локализации ошибок генерируется текст с соответствующим предупреждением.

3.1.3. Стратегия проверки исполнения инструкции `break` на первой итерации цикла

Пусть в цикле, реализующем финитную итерацию над массивом, присутствует операция `break` и эта операция всегда исполняется на первой итерации цикла. Значит, вместо этого цикла можно задать последовательность операций, соответствующих инициализирующему выражению и телу цикла. Такая ситуация может означает наличие ошибки. Поэтому, можно предупредить пользователей системы верификации о такой операции `break`, об условии, при котором исполняется такая операция `break`, и о содержащем такую операцию `break` цикле.

Стратегия проверяет каждый цикл над массивом с инструкцией `break` [121]. Пусть такой цикл встречается i-м в коде программы. Стратегия основана на генерации и проверке истинности следующей леммы:

$$P \to ((j_0 = rep_i(a, args).j) \wedge \\ (rep_i(a, args).loop\text{-}break)),$$

где

- P — предусловие.
- a — массив, над которым исполняется финитная итерация.
- j — счетчик цикла for, реализующего финитную итерацию.
- $args$ — аргументы rep_i, вычисленные с помощью обратного прослеживания. Также вычисляется j_0 – значение счетчика j до исполнения цикла.
- $rep_i(a, args).j$ — значение j после исполнения цикла.

- *loop-break* — специальное поле в возвращаемой структуре данных. Его значение истинно тогда и только тогда, когда при исполнении цикла исполнилась инструкция `break`.

Таким образом, первый конъюнкт заключения леммы является утверждением, что исполнение цикла не изменило значения счетчика цикла. Второй конъюнкт из заключения является утверждением, что при исполнении цикла исполнилась инструкция `break`.

Если лемму удалось доказать, то вычислим условие исполнения `break` с помощью обратного прослеживания. Истинность леммы гарантирует, что итерации, следующие за первой, не исполняются. Отметим, что невозможно вычислить символически условие исполнения `break` на произвольной итерации цикла, так как неизвестно, сколько итераций исполнилось до этой итерации. Но, в случае истинности леммы, возможно символически вычислить условие исполнения `break` с помощью подстановки значения переменных цикла до итерации. Если лемма доказана, то с помощью метода локализации ошибок генерируется текст с предупреждением о такой инструкции `break`, об условии исполнения этой инструкции `break`, и о цикле с этой инструкцией `break`.

3.2. Генерация текстов о сопоставлении конструкций программы и подформул условий корректности

Опишем метод генерации текстов о соответствии подформул УК и исходного кода [19, 114, 121, 126], реализованный в системе C-lightVer. Рассмотрим отличия данного метода от подхода Денни и Фишера [74]:

3.2.1. Язык задания текстовых шаблонов для типов семантических меток

В комплексном подходе используется не ограниченный, а произвольный набор концепций семантических меток [121]. Эта возможность реализована как задание пользователем новых концепций семантических меток и правил вывода

с этими метками [114]. Для каждого типа метки пользователь системы верификации задает шаблон текста [126]. Такие текстовые шаблоны подаются на вход метагенератору [19]. Шаблоны подобны форматным строкам в языке C, так как при их задании можно использовать специальные символы для обозначения диапазона строк. Они задаются для каждой концепции метки с помощью специальной конструкции `label_pattern`, имеющей вид

```
(label_pattern label format_text),
```

где `label` — тип метки, а `format_text` — форматная строка, задающая текстовый шаблон. Он представляет собой описывающий концепцию метки текст на естественном языке, дополненный специальными конструкциями `%begin` и `%end`, на место которых будут подставлены соответственно начало и конец диапазона строк, в котором записан относящийся к данной метке программный код.

В качестве примера рассмотрим задание шаблона текста для метки типа *then*:

```
(label_pattern
    then
    "assumption that \"then\"-branch is chosen at line %begin"
)
```

где `assumption that \"then"\-branch is chosen at line %begin` является текстовым шаблоном для метки типа `then`.

3.2.2. Расширение языка представления правил вывода конструкцией семантических меток

Для поддержки произвольных типов меток в системе C-lightVer язык описания правил вывода был расширен специальной конструкцией *label* [114, 126], используемой для описания меток. Конструкция *label* имеет вид

```
label t c
```

где `t` — терм, снабженный меткой, а `c` — тип метки [19, 113].

В качестве примера рассмотрим задание на языке шаблонов размеченного семантическими метками правила вывода для `while`:

```
{(label P asm_pre)} prog {(label INV ens_inv)},
(label
    {(label INV asm_inv) /\ (label e while_t)} S
        {(label INV ens_inv_iter)}
    pres_inv
),
(label INV asm_inv_exit) /\ (label (not e) while_f) =>
    (label Q ens_post)
|-
{any_predicate(P)} any_code(prog)
{any_predicate(INV)}
while(simple_expression(e)) any_code(S)
{any_predicate(Q)}
```

Данная запись иллюстрирует, что с помощью конструкции `label` можно помечать метками и формулы, и тройки Хоара.

Реализация сопоставления программных конструкций и шаблонов, описанная в разделе 1.7.2, дополнена снабжением термов семантическими метками при применении правил вывода. В ходе работы метагенератора, описанной в разделе, термам сопоставляются характеризующие их метки. Была реализована структура, задающая тип данных метки [18]:

```
struct label{
    char* concept;
    int location_begin;
    int location_end;}
```

В данной структуре поле `concept` содержит тип метки, описывающий предназначение сопоставленного метки терма. Это поле заполняется при синтаксическом анализе правил вывода УК. Поля `location_begin` и `location_end` содержат начало и конец диапазона строк, в котором записан относящийся к данной

метке программный код. Эти поля заполняет ГУК, так как именно при применении аксиом и правил вывода известна информация о позиции программного кода.

В ходе работы ГУК терму, имеющему метку, может быть сопоставлена другая метка и т.д. Эти метки хранятся в соответствующем узле дерева спецификаций `term_node` в виде списка, в котором последним элементом является метка самого верхнего уровня. Данный список представлен в структуре `term_node` атрибутом `labels`.

3.2.3. Семантические метки для функций, выражающих результаты финитных итераций

Так как комплексный подход включает метод семантической разметки, то на вход метагенератору можно подавать размеченные семантическими метками правила вывода [114]. Правило вывода для финитных итераций, описанное в разделе 2.2.4, также было снабжено семантической меткой [126]:

```
{P} prog {(vector_substitution(Q, v,
              (label rep_iter rep(n, v, S, body).vector_element))}
|- {any_predicate(P)} any_code(prog)
for(int_var(i) = 0; int_var(i) < int_var(n); int_var(i)++)
admissible_construct(i, n, v, S, body)
{any_predicate(Q)}
```

Метод метагенерации УК позволяет не переписывать ГУК "вручную" для добавления новых концепций меток [19]. Это позволило удобным образом задать для рассматриваемого правила новый тип метки – *rep_iter*. Меткой с концепцией *rep_iter* снабжается подформула, образованная операцией замены.

Был задан текстовый шаблон для метки `rep_iter` на рассмотренном языке:

```
(label_pattern
    rep_iter
```

```
    "with substitution loop effect from lines %begin-%end by rep"
)
```

где `with substitution loop effect from lines %begin-%end by rep` является текстовым шаблоном для метки типа `rep_iter`.

3.2.4. Алгоритм генерации функций, выражающих результаты финитных итераций, с семантическими метками для итераций над изменяемыми массивами

В комплексном подходе используется добавление семантических меток не только в подформулу УК, но и в определение *rep* [121]. Во-первых, добавление семантических меток в тело функции *rep* позволяет автоматизировать верификацию циклов с помощью символического метода верификации финитных итераций. Во-вторых, добавление семантических меток в тело функции *rep* позволяет сопоставлять финитные операции и определение операции замены для генерации подробных объяснений УК с применениями *rep*.

Рассмотрим случай, когда УК содержит применение операции замены. Рассмотренное правило вывода для операции замены помечает применение функции *rep* в УК специальной семантической меткой. Но такая метка не может предоставить информацию о структуре финитной итерации, которую она помечает. Объяснение УК в таком случае будет содержать текст о том, что применение функции *rep* выражает действие цикла. Но такой текст не содержит информацию о структуре цикла.

Денни и Фишер [74] предложили способ снабжать семантическими метками формулы, но они не предложили способа снабжать семантическими метками определения применяемых в формулах функций. В случае применения операции замены возникла необходимость исправить этот недостаток метода семантической разметки. Поэтому было создано расширение метода семантической разметки для снабжения семантическими метками определения операции замены. Для реализации этого подхода был предложен способ генерации

снабженного метками определения функции rep, способ извлечения меток из определения функции rep и способ генерации текста для списка извлеченных меток. Рассмотрим способ генерации снабженного метками определения функции rep [126].

Для дополнения метками определения функции нужно найти такой способ, который не нарушает семантику языка задания этого определения, и который позволяет упростить извлечение меток из этого определения. Поэтому предложенный способ генерации снабженного метками определения функции rep основан на возможностях, предоставляемых языком системы ACL2. Данный способ представляет собой модификацию алгоритма автоматической генерации операции замены из раздела 2.2.2. Тот алгоритм основан на моделировании последовательности конструкций цикла с помощью последовательности связываний переменной fr с новыми значениями. Но конструкция $b*$ позволяет задать не только последовательное связывание переменных со значением, но и одновременное связывание переменных со значениями. Для одновременного связывания двух переменных с двумя значениями удобно использовать конструкцию $cons$.

Введем переменную $label$, соответствующую семантической метке. Каждое связывание переменной $label$ со значением соответствует своей конструкции цикла и происходит одновременно со связыванием переменной fr с новым значением. Когда мы генерируем связывание, моделирующее очередную конструкцию цикла, мы связываем с новым значением не только переменную fr, но и переменную $label$.

Рассмотрим, что представляет собой значение, связываемое с переменной $label$. Оно представляет собой список следующего вида

$$(list\ 'concept\ begin\ end\ 'break_path),$$

где

- concept – одна из концепций меток. Для помечающих код меток заранее задан определенный набор концепций (типов) меток. Рассмотрим эти тип:

  - *empty_stmt* – тип метки, соответствующей пустому оператору;
  - *break_stmt* – тип метки, соответствующей инструкции `break`;
  - *assign_stmt* – тип метки, соответствующей присваиванию;
  - *if_stmt* – тип метки, соответствующей сокращенному `if`;
  - *full_if_stmt* – тип метки, соответствующей инструкции `if`;
  - *block_stmt* — тип метки, соответствующей блоку;
  - *inner_iter* — тип метки, соответствующей вложенной финитной итерации.

  Отметим, что типы меток однозначно соответствуют видам допустимых конструкций тела финитной итерации;

- begin – начало диапазона строк кода соответствующей метке конструкции;
- end – конец диапазона строк кода соответствующей метке конструкции;
- *'break_path* – опциональный элемент списка, который присутствует в списке тогда и только тогда, когда соответствующая метке конструкция лежит на пути к одному из операторов выхода из цикла.

Такая конструкция представляет собой запись семантической метки.

Также был предложен способ помечать специальной семантической меткой условие инструкции `if`. Для этого выражение e, соответствующее такому условию, записывается как результат конструкции $b*$, но переменная fr связывается только своим значением, так как при вычислении условия инструкции `if` значения переменных не изменяются. В такой конструкции происходит связывание только переменной *label*, а значением этой конструкции является значение e. Переменная *label* связывается в такой конструкции со списком, реализующим семантическую метку. Такой список соответствует введенному ранее формату, но в качестве задающего концепцию первого элемента списка используется *if_cond*.

Модифицированный алгоритм генерации операции замены – это алгоритм из раздела 2.2.2, в котором использование функции *generate_rep* заменено на использование функции *gen_rep*. Эти функции принимают в качестве аргумента допустимую конструкцию и транслируют ее на язык системы ACL2. Функция *c2acl2* осуществляет трансляцию C-kernel выражений на язык системы ACL2.

Определим функцию *gen_rep* для каждого вида допустимой конструкции *op*, обозначив как ... поставляемые номера строк и опциональный элемент *'break_path*:

- если *op* – это пустой оператор, то результатом *gen_rep*(*op*) будет

$$((cons\ ?!label\ fr)\ (cons\ (list\ 'empty_stmt\ \ldots)\ fr));$$

- если *op* — это **break**;, то результатом *gen_rep*(*op*) будет

((cons ?!label fr) (cons
(list 'break_stmt ...)
(change-frame fr :loop-break t)))
((when t) fr);

- если *op* — это **if** (**a**) **b**, то результатом *gen_rep*(*op*) будет

((cons ?!label fr) (cons (list 'if_stmt ...)
(if (b ((cons ?!label ?!fr) (cons*
(list 'if_cond ...) fr)) c2acl2(a))
(b (gen_rep(b)) fr))))*
((when (frame->loop-break fr)) fr);

- если *op* — это **if** (**a**) **b else c**, то результатом *gen_rep*(*op*) будет

((cons ?!label fr) (cons (list 'full_if_stmt ...)
(if (b ((cons ?!label ?!fr) (cons*
(list 'if_cond ...) fr) c2acl2(a))
(b(gen_rep(b)) fr)) (b*(gen_rep(c)) fr))))*
((when (frame->loop-break fr)) fr);

- если *op* — это $\{\mathbf{a_1}\ \mathbf{a_2}\ \dots\ \mathbf{a_{k-1}}\ \mathbf{a_k}\}$, то результатом $gen_rep(op)$ будет

$$((cons\ \ ?!label\ \ fr)\ \ (cons\ (list\ \ 'block_stmt\ \ \dots)$$
$$(b*(gen_rep(a_1)\ gen_rep(a_2)\dots gen_rep(a_{k-1})\ gen_rep(a_k))fr)))$$
$$((when\ \ (frame\text{->}loop\text{-}break\ \ fr))\ \ fr).$$

- если *op* — это $\mathbf{for}\,(\mathbf{j=0;j<m;j++})\ \mathbf{u:=body(u,j)\ end}$, то результатом $gen_rep(op)$ будет

$$(\ \ (cons\ \ ?!label\ \ fr)$$
$$(cons\ \ (list\ \ 'inner_iter\ \ \dots)$$
$$(rep_k\ m\ env_k\ fr_k)))$$

Модифицированный алгоритм генерации операции замены отличается от исходного тем, что он снабжает все виды допустимых конструкций семантическими метками.

3.2.5. Алгоритм генерации объяснений недоказанных условий корректности, содержащих операцию замены

В отличие от подхода Денни и Фишера [74], в комплексном подходе используется извлечение семантических меток из УК не в порядке номеров строк, а в порядке обхода дерева УК в глубину [18]. Это позволяет улучшить генерацию текста в случае вложенных меток [121]. Семантические метки из определения операции замены извлекаются с помощью обхода в глубину дерева кода функции *rep* [126]. В случае применения функции *rep* в УК, дерево кода *rep* рассматривается как поддерево УК и включается в общий обход в глубину для извлечения меток. Опишем метод извлечения семантических меток из специального представления функции *rep* в виде дерева кода.

Представим определение функции *rep* в виде дерева. Построим его по процедуре построения дерева кода функции. Примененная к связываниям, которые соответствуют пустым операторам, инструкциям `break` и присваиваниям тела

цикла, данная процедура возвращает вырожденные деревья в виде соответствующих листьев. Примененная к связыванию, которое соответствует инструкции `if`, данная процедура возвращает дерево, корнем которого является это связывание, а потомки (поддеревья) получены в результате применения данной процедуры к блокам, соответствующим ветвям этой инструкции `if`. Примененная к блоку $b*$, которая соответствует C-блоку, данная процедура возвращает дерево, корнем которого является этот блок, а потомки (поддеревья) получены в результате применения этой процедуры к последовательности связываний, составляющих этот блок. Примененная к вложенной финитной итерации, данная процедура возвращает результат своего применения к вложенной финитной итерации.

Для извлечения меток из определения функции rep для него строится дерево с помощью применения процедуры построения дерева кода функции к блоку $b*$, соответствующему телу цикла. Затем это дерево обходится в глубину, и метки извлекаются из вершин дерева в порядке обхода. Таким образом представление в виде дерева определения функции rep используется для реализации обхода в глубину.

Модифицированный алгоритм генерации объяснений недоказанных условий корректности – это алгоритм из раздела 2.2.2, дополненный способом генерации текста для меток, извлеченных из специального представления функции rep в виде дерева кода. Опишем способ генерации текста для меток, извлеченных из определения функции rep.

В результате извлечения меток из УК, получается упорядоченный список меток.

В отличие от подхода Денни и Фишера [74], в комплексном подходе используются заданные пользователем шаблоны объяснений меток для генерации объяснений всего УК [18].

При генерации текста список извлеченных меток обрабатывается последовательно. Для каждой метки происходит извлечение диапазона строк, затем с помощью процедуры, позволяющей вернуться от конструкций промежуточной C-kernel программы к конструкциям исходной C-light программы, данный

диапазон строк C-kernel программы переводится в диапазон строк C-light программы. Потом для данной метки генерируется текст с помощью подстановки полученных номеров строк C-light программы. Таким образом, общий текст получается из набора текстов для каждой метки.

Если в верифицируемой программе содержится финитная итерация с инструкцией `break`, то генерируется дополнительный поясняющий текст. УК такой программы представляет собой импликацию, содержащую применение функции rep. Рассмотрим посылку такого УК. Проанализируем случай, описываемый рассматриваемой посылкой, используя результат применения стратегии для программ с выходом из цикла из раздела 2.5.2.

Если удалось доказать лемму, что в описываемом рассматриваемой посылкой случае исполнилась инструкция `break`, то в данном случае рассмотрим связывания, соответствующие инструкциям `if` на пути к данной инструкции. Именно для этой цели список, соответствующий метке, содержит опциональный элемент $'break_path$. Используя все условия условных операторов, вычислим символически с помощью подстановок общее условие, при выполнении которого исполняется связывание, соответствующее инструкции `break`. Обозначим такое условие как θ. Так как в рассматриваемом случае инструкция `break` исполнилась, то хотя бы для одной итерации условие θ истинно. Значит, пользователю системы верификации имеет смысл проверить условие θ на предмет наличия ошибки. Поэтому об этом генерируется дополнительный поясняющий текст [126]. Этот текст содержит информацию об исполнении инструкции $break$, заданной на определенной строке программы, содержит информацию о существовании такого номера итерации, что условие θ выполняется на этой итерации, и содержит условие θ.

После этого последовательно запускается исполнение всех стратегий локализации ошибок в случае недоказанного УК [121]. Если УК не удалось доказать, то к циклам, соответствующим операциям замены из этого УК, последовательно применяются стратегии:

- Стратегия проверки ложности недоказанных УК.

- Стратегия поиска циклов с неиспользуемыми присваиваниями элементам массива.

- Стратегия проверки исполнения инструкции `break` на первой итерации цикла.

Если применение стратегии локализации ошибок позволяет доказать ложность УК или выполнение соответствующего свойства для цикла, то, можно предположить, что программа содержит ошибку. В таком случае модуль локализации ошибок запускает генерацию текста с предупреждением о возможной ошибке, используя шаблон для стратегии. Номера строк и выражения из программы подставляются в специальные места шаблона. Полученный текст добавляется к объяснению УК. В итоге, получается отчет о локализации ошибок. Примеры отчетов о локализации ошибок можно посмотреть в разделах 4.2.2, 4.3.2, 4.4.1 и 4.4.3.

3.3. Выводы

На основе сделанного описания метода локализации ошибок можно сделать следующие выводы:

- Дополнительная информация, которую возвращают функции, выражающие результат финитных итераций, позволяет генерировать леммы о свойствах финитных итераций не только стратегиям доказательства УК, но и стратегиям локализации ошибок. Отметим, что модернизированный алгоритм генерации таких функций не только сохраняет дополнительную информацию в возвращаемом значении, но и снабжает код данных функций семантическими метками. Таким образом, комплексный подход предоставляет единый способ анализировать вопрос корректности и некорректности программ, используя информацию, сохраненную в определениях функций rep и их возвращаемых значениях.

- Метод метагенерации УК позволил расширить систему C-lightVer возможностью задавать произвольные типы семантических меток. После реализации в метагенераторе анализа предложенных языков, задание новых типов семантических меток и разметка ими правил вывода не требует внесения изменений в реализацию метагенератора УК. Таким образом, продемонстрированы преимущества комплексного подхода к автоматизации дедуктивной верификации.

Рассмотрим в следующих главах применение методов комплексного подхода к программам на языках Cloud Sisal, C и расширении языка C циклами языка Cloud Sisal.

Глава 4

Применение разработанных методов для дедуктивной верификации С-программ с финитными итерациями

Опишем применение комплексного подхода к программам на языке C [117, 118, 121, 126]. Рассмотрим созданную в результате работы, описанной в данной диссертации, модифицированную систему C-lightVer [17, 19, 114, 121]. Опишем эксперименты по верификации программ с финитными итерациями на языке C-light с помощью модифицированной системы C-lightVer [117, 118, 121, 144]. Продемонстрируем использование комплексного подхода для избежания задания инвариантов циклов, автоматизации доказательства УК и автоматизации локализации ошибок [114, 117, 121, 126].

4.1. Модифицированная версия системы C-lightVer

Модифицированная версия системы C-lightVer отличается от исходной реализацией методов комплексного подхода. Данная реализация привела к модификации всех модулей системы и созданию модулей управления доказательством УК, генерации лемм и локализации ошибок. Был разработан прототип модифицированной системы C-lightVer для дедуктивной верификации C-программ [17, 19, 114, 117, 118, 121, 126]. Рассмотрим модифицированную версию системы C-lightVer. Схема системы C-lightVer изображена на рис. 4.1. Система состоит из семи основных модулей:

1. Модуль трансляции из C-light в C-kernel принимает на вход аннотированную C-light программу и производит
    - трансляцию в эквивалентную аннотированную C-kernel программу;

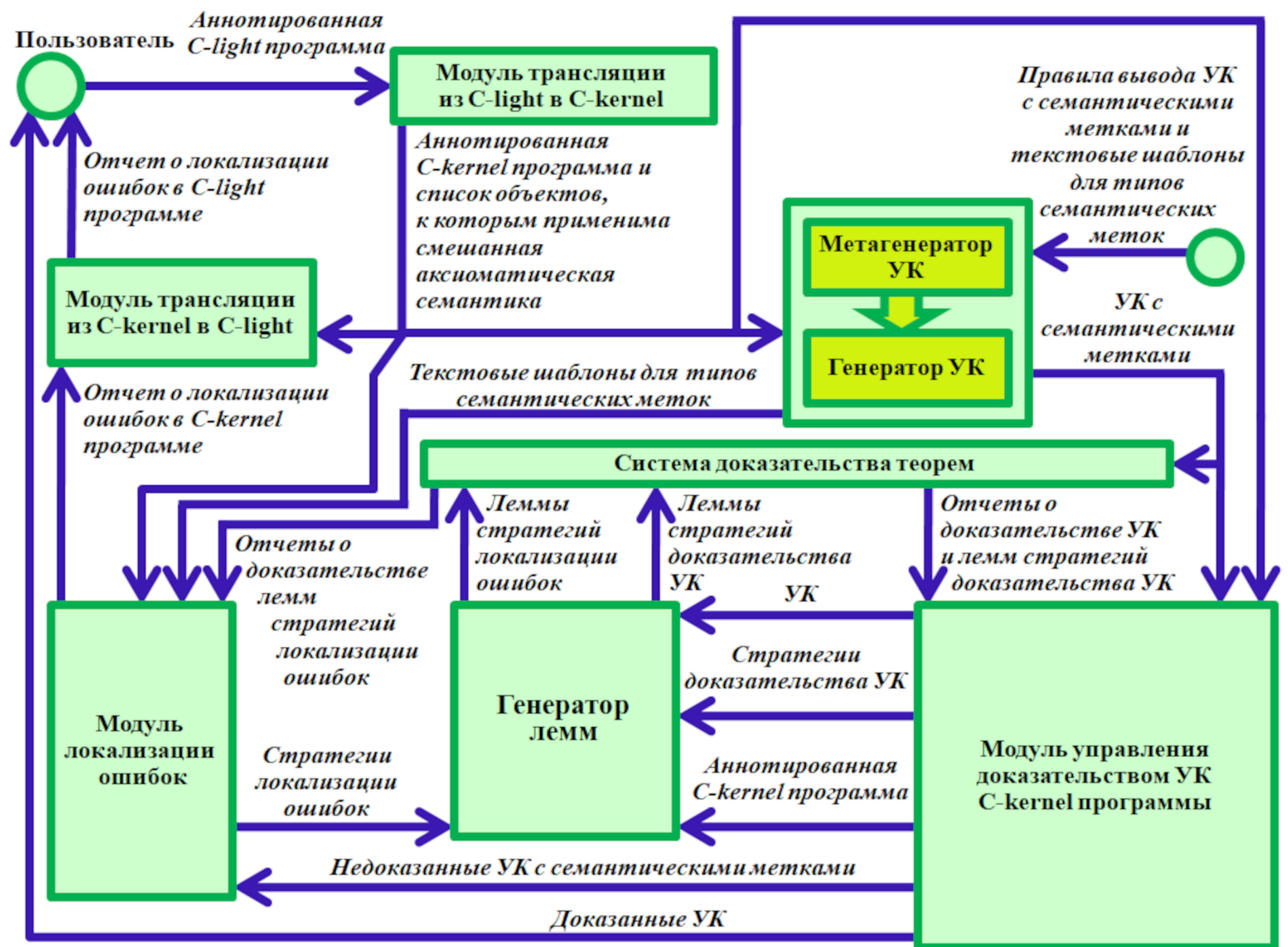

Рис. 4.1. Модифицированная версия системы C-lightVer

- поиск объектов, к которым применима смешанная аксиоматическая семантика;
- добавление информации [147], позволяющей транслировать конструкции C-kernel программы в конструкции C-light программы.

Эта информация добавляется к конструкциям, измененным в результате трансляции. Это позволяет снабдить конструкции названиями примененных правил трансляции.

2. Модуль метагенерации УК принимает на вход аннотированное промежуточное представление программы, правила вывода с семантической разметкой на языке задания шаблонов и текстовые шаблоны для типов семантических меток. Модуль порождает формулы, из истинности которых следует корректность программы. Если программа содержит финитные

итерации, то данный модуль генерирует операции замены для них.

3. Модуль управления доказательством последовательно применяет все подходящие стратегии для доказательства УК. Их применение состоит в генерации и доказательстве лемм. Доказанные леммы добавляются в теорию предметной области, которая используется системами доказательства теорем. Если применение стратегий привело к доказательству всех УК, то C-lightVer завершает исполнение.

4. Модуль генерации лемм принимает на вход либо стратегию доказательства УК, либо стратегию локализации ошибок. Данный модуль генерирует леммы, соответствующие входным стратегиям. Леммы передаются на вход модулю доказательства теорем.

5. Модуль доказательства теорем проверяет истинность УК и лемм. В качестве данного модуля используются системы доказательства теорем.

6. Модуль локализации ошибок последовательно применяет все подходящие стратегии локализации ошибок в случае недоказанного УК. Их применение состоит в генерации и доказательстве лемм. Данный модуль также генерирует отчет о локализации ошибок. Данный модуль генерирует объяснение недоказанных УК, используя семантические метки с информацией о соответствии между подформулами УК и конструкциями программы. Это является базовой частью отчета. Если система доказательства теорем доказала лемму, порожденную стратегией локализации ошибок, то в отчет добавляется текст, представляющий эту стратегию.

7. Модуль трансляции конструкций C-kernel в конструкции C-light основан на использовании информации, добавленной транслятором из C-light в C-kernel. Эта информация позволяет применить специальные правила трансляции [147] из C-kernel в C-light. Отчет о результатах верификации, который генерирует модуль локализации ошибок, основан на информации, хранящейся в семантических метках. Но метки содержат информа-

цию не о конструкциях C-light программы, а о конструкциях C-kernel программы. Данный модуль заменяет конструкции C-kernel программы на конструкции C-light программы в отчете о локализации ошибок. Также номера строк C-kernel программы заменяются на номера строк C-light программы. Полученный отчет о локализации ошибок в C-light программе передается пользователю.

Модули пронумерованы в соответствии с этапами исполнения модифицированной версии системы C-lightVer.

4.2. Верификация программ с финитными итерациями над неизменяемыми массивами без инструкции `break`

Рассмотрим эксперименты по применению комплексного подхода к C-программам с финитными итерациями, которые относятся к классу итераций над неизменяемыми массивами без инструкции `break` [19, 114].

4.2.1. Сумма абсолютных значений элементов вектора

Отметим, что представляет применение символического метода верификации финитных итераций к программам, реализующим интерфейс BLAS [77] операций линейной алгебры. Поэтому, рассмотрим дедуктивную верификацию программы `asum` [117]:

```
int abs_sum(int *a, int n){
    int asum = 0;
    for (int i = 0; i < n; i++){
        if (a[i] < 0) asum += -a[i];
        else asum += a[i];}
    return asum;}
```

Данная функция вычисляет сумму абсолютных значений вектора `a`.

Приведем предусловие на языке системы ACL2:

```
(and (integer-listp a) (integerp n) (0 < n) (<= n length(a)))
```

Постусловие на языке системы ACL2 выглядит следующим образом:

```
(= (abs-sum 0 (- n 1) a) asum)
```

где `abs-sum` — функция из теории предметной области, вычисляющая сумму абсолютных значений третьего аргумента в диапазоне от первого аргумента до второго аргумента включительно. Приведем определение функции `abs-sum` на языке системы ACL2:

```
(defun abs-sum(i j a)
    (if (or (not (natp i)) (not (natp j))) 0
        (if (> i j) 0
            (if (= i j) (abs (nth j a))
                (+ (abs (nth j a)) (abs-sum i (- j 1) a))))))
```

Определение функции *rep* было сгенерировано с помощью алгоритма, описанного в разделе 2.2.2. Данное определение на языке системы ACL2 выглядит следующим образом:

```
(define rep ((i natp)(fr frame-p))
  (b* (((when (zp i)) (frame-fix fr))
       (fr (rep (- i 1) fr))
       (fr (if
             (< (nth (- i 1) (frame->a fr)) 0)
             (b*
                 ((fr (change-frame fr :asum
                  (+ (frame->asum fr) (- (nth (- i 1) (frame->a fr))))))) fr)
             (b*
                 ((fr (change-frame fr :asum
                  (+ (frame->asum fr) (nth (- i 1) (frame->a fr)))))) fr)))) fr)
```

Было сгенерировано следующее УК:

```
(defrule vc-1
   (implies
    (and (integer-listp a) (integerp n) (< 0 n) (<= n (length a)))
    (equal (abs-sum 0 (- n 1) a) (frame->asum (rep n (frame-init a 0)))))
      :induct (dec-induct n))
```

Система ACL2 автоматически доказала данное УК с помощью индукции по n.

4.2.2. Скалярное произведение векторов

Так как интерес представляет применение символического метода верификации финитных итераций к программам, реализующим интерфейс BLAS [77] операций линейной алгебры, то рассмотрим дедуктивную верификацию функции `dot_product` [114]:

```
/*@ requires (x != NULL) && (y != NULL) && (length > 0);
    ensures (value = DP_RESULT(0, length, x, y)); */
int dot_product(unsigned int length, int* x, int* y){
    int value = 0;
    for (unsigned int i = 0; i < length; i++){
        value += x[i] * y[i];}
    return value;}
```

Данная функция вычисляет скалярное произведение векторов `x` и `y`. Значение скалярного произведения накапливается в переменной `value`. Для финитной итерации из функции `dot_product` вектор изменяемых переменных v состоит из переменной `value`.

В качестве языка спецификаций данной функции используется ACSL. Постусловие основано на использовании функции `DP_RESULT` из теории предметной области.

Рассмотрим сначала первый эксперимент по дедуктивной верификации функции `dot_product` [114]. В качестве системы доказательства теорем в данном эксперименте был использован SMT-решатель Z3. Приведем определение функции `DP_RESULT` на языке SMT-решателя Z3:

```
(declare-fun DP_RESULT (Int Int (Array Int Int) (Array Int Int)) Int)
(assert (and
    (forall ((i Int) (j Int) (x (Array Int Int)) (y (Array Int Int)))
        (implies (<= j i) (= (DP_RESULT i j x y) 0)))
```

```
(forall ((i Int) (j Int) (x (Array Int Int)) (y (Array Int Int)))
    (implies (< i j) (= (DP_RESULT i j x y) (+
(DP_RESULT i (- j 1) x y) (* (select x (- j 1)) (select y (- j 1))))))))
```

Эта функция определена рекурсивно и вычисляет скалярное произведение двух векторов от i-го до j-го элемента.

Определение функции *rep* было сгенерировано с помощью алгоритма для генерации операции замены над неизменяемыми массивами, описанного в разделе 2.1. Была получена функция *rep*, соответствующая переменной *value*. Рассмотрим сгенерированные на языке SMT-решателя Z3 аксиомы, определяющие функцию *rep*:

```
(declare-fun rep (Int Int (Array Int Int) (Array Int Int)) Int)
(assert (and
    (forall ((i Int) (value Int) (x (Array Int Int)) (y (Array Int Int)))
        (implies (<= i 0) (= (rep i value x y) value)))
    (forall ((i Int) (value Int) (x (Array Int Int)) (y (Array Int Int)))
        (implies (< 0 i) (= (rep i value x y) (+
  (rep (- i 1) value x y) (* (select x (- i 1)) (select y (- i 1))))))))
```

Z3 является SMT-решателем, но в данном эксперименте проверяется истинность УК, а не выполнимость. Поэтому, ГУК генерирует отрицание УК:

```
(assert (not (forall ((value Int) (length Int)
                      (x (Array Int Int)) (y (Array Int Int)))
    (implies (> length 0) (= (rep length 0 x y) (DP_RESULT 0 length x y))))))
```

Ожидаемым ответом является “unsat”, который означает, что отрицание невыполнимо, то есть УК истинно.

Однако, в данном случае результатом исполнения Z3 является “unknown”. Это означает, что Z3 не может определить, выполнима ли формула или нет. Был применен метод Лейно, который предложил для SMT-решателей отдельно доказывать шаг индукции и индукционный переход [138]. SMT-решатель Z3 доказал базу индукции, но не доказал индукционный переход. Отметим, что Z3

доказывает УК в случае конечных длин от массива от `1` до `19` включительно (значение переменной `length`). Но, в итоге, УК не было доказано.

Так как УК не было доказано, то с помощью алгоритма генерации объяснений УК, описанном в разделе 3.2.5, было получено следующее объяснение (в приведенном выше УК были опущены семантические метки):

```
This formula corresponds to lines 6-10
in function "dot_product".
Hence, given
    - assumption that precondition
      from line 1 holds,
ensure that postcondition goal from line 2
    with substitution loop effect
    from lines 5-6 by rep
```

Данный текст на естественном языке объясняет не только структуру, но и цель УК.

Теперь рассмотрим второй эксперимент по дедуктивной верификации функции `dot_product` [114]. В качестве системы доказательства теорем в данном эксперименте была использована система PVS. Определение функции *rep* было сгенерировано с помощью алгоритма для генерации операции замены над неизменяемыми массивами, описанного в разделе 2.1, дополненного генерацией меры как первого аргумента функции *rep* (номера итерации). Система PVS требует задания меры при определении рекурсивных функций для доказательства их завершимости. Была получена следующая теория предметной области, заданная на языке системы PVS:

```
dot_product: THEORY
BEGIN
  DP_RESULT(i:nat, j:nat, x:ARRAY[nat->int], y:ARRAY[nat->int]): RECURSIVE int =
     IF j <= i THEN 0
     ELSE x(j-1) * y(j-1) + DP_RESULT(i, j-1, x, y) ENDIF
  MEASURE j
  rep(i:nat, value:int, x:ARRAY[nat->int], y:ARRAY[nat->int]): RECURSIVE int =
     IF i <= 0 THEN value
```

```
    ELSE x(i-1)*y(i-1)+rep(i-1, value, x, y) ENDIF
  MEASURE i
  vc: LEMMA
     FORALL (value:int, length:nat, x:ARRAY[nat -> int], y:ARRAY[nat -> int]):
     (0 < length) IMPLIES
     rep(length, 0, x, y) = DP_RESULT(0, length, x, y)
END dot_product
```

Эта теория содержит определения функций `DP_RESULT`, `rep` и условие корректности. Система PVS предоставляет набор тактик доказательства по индукции. В данном эксперименте была применена следующая PVS-тактика:

```
(induct-and-simplify "length")
```

Данная тактика запускает доказательство, основанное на индукции по переменной-аргументу тактики и на автоматическом упрощении базы индукции и индукционного перехода. Данная тактика была применена к переменной `length` при доказательстве УК. В результате был получен следующий ответ от системы PVS:

```
By induction on length, and by repeatedly rewriting and simplifying,
Q.E.D.
Run time  = 0.14 secs.
Real time = 19.32 secs.
```

Таким образом, применение данной тактики позволило доказать УК в автоматическом режиме.

4.3. Верификация программ с финитными итерациями над неизменяемыми массивами с инструкциями `break`

Рассмотрим эксперименты по применению комплексного подхода к C-программам с финитными итерациями, которые относятся к классу итераций над неизменяемыми массивами с инструкциями `break` [126, 144].

4.3.1. Проверка наличия в массиве не менее заданного количества вхождений ключа

Рассмотрим эксперимент по дедуктивной верификации функции **search_count** [144]. Для заданных целых чисел key и $entr$ эта функция возвращает 1, если не менее чем $entr$ элементов данного массива целых чисел arr равны key, где $length$ — это длина arr. Иначе, эта функция возвращает 0. Рассмотрим данную аннотированную функцию на языке C-light:

```
/* (assert (and (> length 0) (> entr 0))) */
int search_count(int* arr, int length, int key, int entr){
    auto int result = 0, cnt = 0, i;
    for (i = 0; i < length; i++){
        if ((key == arr[i]) && (entr > (cnt + 1))){
            cnt++;}
        else if ((key == arr[i]) && (entr == (cnt + 1))){
            cnt++; result = 1; break;}}
    return result;
}
/* (and
       (implies (<= entr (COUNT length arr key entr)) (= result 1))
       (implies (> entr (COUNT length arr key entr)) (= result 0)) */
```

В данном эксперименте в качестве системы доказательства был использован SMT-решатель CVC4. Функция **COUNT** возвращает количество вхождений key в arr от 0 to $length - 1$. Рассмотрим аксиомы, задающие определение функции **COUNT** на языке системы CVC4:

```
(declare-fun COUNT (Int (Array Int Int) Int Int) Int)
(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
   (=> (and (< 0 i) (= (select arr (- i 1)) key))
       (= (COUNT i arr key entr) (+ (COUNT (- i 1) arr key entr) 1)))))
(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
   (=> (and (< 0 i) (not (= (select arr (- i 1)) key)))
       (= (COUNT i arr key entr) (COUNT (- i 1) arr key entr)))))
```

Для финитной итерации из функции `search_count` вектор изменяемых переменных v равен $(cnt, result)$ и его изначальными значениями являются $(0, 0)$. Определения функций rep были сгенерированы с помощью алгоритма для генерации операции замены над неизменяемыми массивами с выходом из цикла, описанного в разделе 2.1. Была получена функция rep_1, соответствующая переменной cnt, и функция rep_2, соответствующая переменной $result$. Рассмотрим сгенерированные на языке системы CVC4 аксиомы, определяющие функции rep_1 и rep_2:

```
(declare-fun rep1 (Int (Array Int Int) Int Int) Int)
(declare-fun rep2 (Int (Array Int Int) Int Int) Int)
(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
    (=> (and (< 0 i) (= key (select arr (- i 1)))
             (> entr (+ (rep1 (- i 1) arr key entr) 1)))
        (= (rep1 i arr key entr) (+ (rep1 (- i 1) arr key entr) 1)))))
(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
    (=> (and (< 0 i) (not (= key (select arr (- i 1))))
             (> entr (+ (rep1 (- i 1) arr key entr) 1)))
         (= (rep1 i arr key entr) (rep1 (- i 1) arr key entr)))))
(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
    (=> (and (< 0 i) (> entr (+ (rep1 (- i 1) arr key entr) 1)))
         (= (rep2 i arr key entr) (rep2 (- i 1) arr key entr)))))
(assert (forall ((i Int) (arr (Array Int Int)) (key Int) (entr Int))
    (=> (and (< 0 i) (= key (select arr (- i 1)))
             (= entr (+ (rep1 (- i 1) arr key entr) 1)))
         (= (rep2 i arr key entr) 1))))
```

CVC4 является SMT-решателем, но в данном эксперименте проверяется истинность УК, а не выполнимость. Поэтому, ГУК генерирует отрицание УК:

```
(assert (not (forall ((length Int) (arr (Array Int Int))
                      (key Int) (entr Int) (result Int))
    (=> (and (> length 0) (> entr 0) (= result (rep2 j arr key entr)))
        (and (=> (<= entr (COUNT length arr key entr))
                 (= result 1))
             (=> (> entr (COUNT length arr key entr))
                 (= result 0)))))))
```

Ожидаемым ответом является “unsat”, который означает, что отрицание невыполнимо, то есть УК истинно.

Но в данном случае результатом исполнения CVC4 является “unknown”. Это означает, что CVC4 не может определить, выполнима ли формула или нет. Рассмотрим применение стратегий доказательства к данному УК.

Для доказательства представим УК в виде двух конюъанктов. Рассмотрим первый такой конъюнкт:

```
(assert (forall ((length Int) (arr (Array Int Int))
                 (key Int) (entr Int) (result Int))
                 (=> (and (> length 0) (> entr 0)
                          (= result (rep2 length arr key entr))
                          (> entr (COUNT length arr key entr)))
   (= result 0))))
```

и второй такой конъюнкт:

```
(assert (forall ((length Int) (arr (Array Int Int))
                 (key Int) (entr Int) (result Int))
                (=> (and (> length 0) (> entr 0)
                         (= result (rep2 length arr key entr))
                         (> entr (COUNT length arr key entr)))
                   (= result 0))))
```

Назовем их соответственно первой и второй частью условия корректности.

Применим стратегию интерактивного доказательства, описанную в разделе 2.4.1, к первому конъюнкту. Для применения алгоритма добавим к набору аксиом следующую теорему:

```
(assert (forall ((j Int) (arr (Array Int Int))
                 (key Int) (entr Int))
                 (=> (and (> j 0) (> entr 0)
                     (> entr (COUNT j arr key entr)))
                    (> entr (rep1 j arr key entr)))))
```

Истинность данной теоремы успешно доказывается при использовании CVC4 с помощью метода Лейно, который предложил для SMT-решателей отдельно доказывать шаг индукции и индукционный переход [138].

Конюънкты из посылки данной формулы совпадают с некоторыми конюънктами первой части условия корректности с точностью до переименования переменных. Поэтому, заключение этой формулы добавляется в качестве конъюнкта в посылке исходной формулы на шаге 7 стратегии интерактивного доказательства. Для доказательства измененной таким образом первой части условия корректности применим метод Лейно. База индукции доказывается достаточно просто. Для доказательства индукционного шага проверим невыполнимость его отрицания:

```
(assert (not (forall ((length Int))
     (=> (forall ((arr (Array Int Int)) (key Int)
                  (entr Int) (result Int))
                 (=> (and (> length 0) (> entr 0)
                          (= result (rep2 length arr key entr))
                          (> entr (COUNT length arr key entr))
                          (> entr (rep1 length arr key entr)))
                    (= result 0)))
           (forall ((arr (Array Int Int)) (key Int)
                    (entr Int) (result Int))
                 (=> (and (> (+ length 1) 0) (> entr 0)
                          (= result (rep2 (+ length 1) arr key entr))
                          (> entr (COUNT (+ length 1) arr key entr))
                          (> entr (rep1 (+ length 1) arr key entr))
                     (= result 0)))))))
```

Истинность данной теоремы успешно проверяется с помощью CVC4 (проверяется невыполнимость отрицания теоремы). Таким образом доказана первая часть УК.

Вторая часть УК доказывается аналогично первой. Для применения стратегии интерактивного доказательства добавим к набору аксиом следующую теорему:

```
(assert (forall ((j Int) (arr (Array Int Int))
                 (key Int) (entr Int))
                 (=> (and (< 0 j) (< 0 entr)
                          (> entr (COUNT j arr key entr)))
                  (> entr (rep1 j arr key entr)))))
```

Истинность данной теоремы доказывается по индукции с помощью метода Лейно при использовании CVC4. Также на шаге 7 стратегии интерактивного доказательства для успешного доказательства применяется метод Лейно. Таким образом доказывается вторая часть УК.

Значит, условие корректности доказано.

4.3.2. Проверка наличия в массиве элемента, большего или равного заданному ключу

Рассмотрим функцию `grt_eql_key` [126]:

```
1. /*@ requires (0 < n) && (n <= len(a));
2.     ensures (grt-eql-cnt(n, key, a) == 0 ==>
                                 \result == 0) &&
3.             (grt-eql-cnt(n, key, a) > 0 ==>
                                 \result == 1)
4. */
5. int grt_eql_key(int n, int key, int a[]){
6.    int i, result = 0;
7.    for (i = 0 ; i < n; i++){
8.        if (a[i] >= key){result = 1; break;}}
9.    return result;}
```

Спецификации функции заданы на языке ACSL. Определение функции *grt-eql-cnt* может быть найдено в репозитории [112].

В соответствии со спецификацией функция `grt_eql_key` проверяет существование в массиве `a` элемента, который больше ключа `key` или равен ключу `key`. Если такой элемент найден, то по спецификации функция должна возвращать `1`, иначе – `0`.

Сначала рассмотрим первый эксперимент по автоматизированной локализации ошибки в функции *grt-eql-cnt* [126]. Рассмотрим функцию `grt_eql_key` с внесенной ошибкой:

```
1. /*@ requires (0 < n) && (n <= len(a));
2.     ensures (grt-eql-cnt(n, key, a) == 0 ==> \result == 0) &&
3.             (grt-eql-cnt(n, key, a) > 0 ==> \result == 1)
4. */
5. int grt_eql_key(int n, int key, int a[]){
6.    int i, result = 0;
7.    for (i = 0 ; i < n; i++){
8.        if (a[i] < key){result = 1; break;}}
9.    return result;}
```

Ошибка состоит в использовании оператора `<` вместо оператора `>=` в инструкции `if` тела цикла. Дополнительные материалы по данному эксперименту приведены в репозитории [112].

Приведем результат трансляции программы на язык C-kernel:

```
 5. int grt_eql_key(int n, int key, int a[]){
 6.    /* begin changes Dec3 1 7-8 */
 7.    auto int i;
 8.    auto result = 0;
 9     /* end changes */
10.    for (i = 0 ; i < n; i++){
11.        if (a[i] < key){result = 1; break;}}
12.    return result;}
```

Отметим, ни спецификация, ни финитная итерация не изменяется транслятором. Строка `begin changes Dec3 1 7-8` хранит информацию, используемую процедурой, позволяющей от конструкций промежуточной C-kernel программы вернуться к конструкциям исходной C-light программы. Эта информация утверждает, что было применено правило трансляции деклараций `Dec3` и результат применения этого правила записан в строках `7-8`. Для данной программы с помощью правила вывода, описанного в разделе 3.2.3, было сгенерировано следующее УК (формула vc-1):

$$\forall n, key, a$$
$$((\ulcorner 0 < n \wedge n \leq len(a)^{\urcorner ass_pre(1)} \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge$$
$$\ulcorner grt\text{-}eql\text{-}cnt(n, key, a) = 0^{\urcorner ass_post(2)} \rightarrow$$
$$\ulcorner\ulcorner rep(n, key, a, 0).result^{\urcorner rep_iter(10-11)} = 0^{\urcorner ens_post(2)}) \wedge$$
$$(\ulcorner 0 < n \wedge n \leq len(a)^{\urcorner ass_pre(1)} \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge$$
$$\ulcorner grt\text{-}eql\text{-}cnt(n, key, a) > 0^{\urcorner ass_post(3)} \rightarrow$$
$$\ulcorner\ulcorner rep(n, key, a, 0).result^{\urcorner rep_iter(10-11)} = 1^{\urcorner ens_post(3)})$$

где Int – множество целых чисел, $IntArr$ – множество целочисленных массивов. Термы *vc*-1 помечены семантическими метками с диапазонами строк. Концепция метки ass_pre означает гипотезу из предусловия, концепция метки ass_pre означает гипотезу из постусловия, концепция метки ens_post означает цель из постусловия, концепция метки rep_iter означает уточнение о подстановке операции замены.

Данное УК основано на функции *rep*. Сначала определение функции *rep* было сгенерировано без семантических меток с помощью алгоритма, описанного в разделе 2.2.2. Приведем данное определение:

```
(define rep ((iteration natp) (env envir-p) (fr frame-p))
    (b* (   (iteration (nfix iteration)) (env (envir-fix env))
            (fr (frame-fix fr)) ((when (zp iteration)) fr)
            (fr (rep (- iteration 1) env fr)) ((when (frame->loop-break fr)) fr)
            (fr (if (< (nth (- (+ iteration (envir->lower-bound env)) 1)
                            (envir->a env)) (envir->key env))
                    (b* (   (fr (change-frame fr :result 1))
                            (fr (change-frame fr :loop-break t))
                            ((when t) fr)) fr)
                    (b* ( (fr fr)) fr)))
            ((when (frame->loop-break fr)) fr)
            (fr (change-frame fr :i
                     (+ iteration (envir->lower-bound env))))) fr))
```

Формулу *vc*-1 не удалось доказать с помощью стратегий доказательства УК. Поэтому, к данному УК была применена стратегия проверки ложности недоказанных условий корректности, описанная в разделе 3.1.1.

Формула vc-1 является аргументом ω стратегии проверки ложности недоказанных УК. Формула vc-1 является конъюнкцией двух импликаций. Заключение каждой импликации основано на использовании функции rep. Поэтому, для первого конъюнкта ϕ была сгенерирована формула U_1:

$$\exists n, key, a$$
$$0 < n \land n \leq len(a) \land n \in Int \land key \in Int \land a \in IntArr \land grt\text{-}eql\text{-}cnt(n, key, a) = 0$$

Так как пользователь системы верификации знает определение функции $grt\text{-}eql\text{-}cnt$ из постусловия программы, то пользователь сообщил об истинности U_1. Следовательно, для первого конъюнкта ϕ была сгенерирована формула V_1:

$$\forall n, key, a$$
$$\ulcorner 0 < n \land n \leq len(a) \urcorner^{ass_pre(1)} \land n \in Int \land key \in Int \land a \in IntArr \land$$
$$\ulcorner grt\text{-}eql\text{-}cnt(n, key, a) = 0 \urcorner^{ass_post(2)} \rightarrow$$
$$\ulcorner \ulcorner rep(n, key, a, 0).result \urcorner^{rep_iter(10-11)} \neq 0 \urcorner^{ens_post(2)}$$

Так как пользователь не знает определения функции rep, то необходимо доказывать данную формулу в автоматическом режиме. Также трудности для доказательства вызваны оператором `break` в теле цикла. Поэтому, для доказательства формулы V_1 была использована стратегия для программ с выходом из цикла, описанная в разделе 2.5.2. Определение леммы V_1 на языке системы ACL2 может быть найдено в репозитории [112].

Для V_1 была сгенерирована формула V_1-$lemma$-1:

$$\forall n, key, a$$
$$0 < n \land n \leq len(a) \land n \in Int \land key \in Int \land a \in IntArr \land grt\text{-}eql\text{-}cnt(n, key, a) = 0 \rightarrow$$
$$rep(n, key, a, 0).loop\text{-}break$$

Объясним данную лемму. Так как $0 < n$, то тело цикла исполнится как минимум один раз. Количество элементов массива a, которые больше или равны ключу key, равно нулю. Поэтому, все элементы массива a в диапазоне $[0 : n-1]$ меньше ключа key. Так как в условии оператора if используется неправильный знак, то для такого массива оператор $break$ должен сработать. Определение леммы V_1-$lemma$-1 на языке системы ACL2 может быть найдено в репозитории [112].

Лемма V_1-*lemma*-1 была автоматически доказана индукцией по n в системе ACL2 и добавлена в предметную область.

Срабатывание оператора `break` означает, что значение *result* равно 1. Поэтому, использование леммы V_1-*lemma*-1 позволило доказать формулу V_1 в автоматическом режиме. Значит, формула *vc*-1 ложна. Так как использование первого конъюнкта УК позволило доказать ложность формулы *vc*-1, то было сгенерировано объяснение формулы V_1. Приведем сгенерированный отчет о локализации ошибок:

```
This formula corresponds to lines 1-9 in function "grt_eql_key".
Its purpose is to show unsatisfiable case. Hence, given
    - assumption that precondition from line 1 holds,
    - assumption that postcondition hypothesis from line 2 holds,
ensure that postcondition goal from line 2
    with substitution loop effect from lines 7-8 by rep
does not hold
```

Данное объяснение было сгенерировано автоматически модулем локализации ошибок, используя анализ семантических меток формулы V_1 и анализ результатов применения стратегии доказательства ложности. Но в метках хранятся диапазоны строк C-kernel программы. Информация о применении правил трансляции позволила в объяснении использовать номера строк исходной C-light программы. Так как данный текст описывает ошибочный случай, то данное объяснение может помочь пользователю понять местоположение ошибки.

Рассмотрим второй эксперимент по автоматизированной локализации ошибки в функции *grt-eql-cnt* [126]. Данный эксперимент отличается от первого эксперимента только использованием для генерации определения функции *rep* другого алгоритма, описанный в разделе 3.2.4. Этот алгоритм позволяет снабдить определение функции *rep* семантическими метками. Это позволяет сгенерировать более подробное объяснение формулы V_1. Приведем определение функции *rep* с семантическими метками на языке системы ACL2:

```
(define rep ((iteration natp) (env envir-p) (fr frame-p))
    (b* (   (iteration (nfix iteration)) (env (envir-fix env))
```

```
(fr (frame-fix fr)) ((when (zp iteration)) fr)
(fr (rep (- iteration 1) env fr))
((when (frame->loop-break fr)) fr)
((cons ?!label fr) (cons (list 'if_stmt 6 8 'break_path)
        (if (b* ( ((cons ?!label ?!fr)
                        (cons (list 'if_cond 6 6 'break_path) fr)))
                (> (envir->key env)
                   (nth (- (+ iteration (envir->lower-bound env)) 1)
                        (envir->a env))))
            (b* ( ((cons ?!label fr)
                        (cons (list 'assign_stmt 7 7 'break_path)
                            (change-frame fr :result 1)))
                  ((cons ?!label fr)
                        (cons (list 'break_stmt 8 8 'break_path)
                            (change-frame fr :loop-break t)))
                  ((when t) fr)) fr)
            (b* ( (fr fr)) fr))))
((when (frame->loop-break fr)) fr)
(fr (change-frame fr :i
         (+ iteration (envir->lower-bound env))))) fr))
```

Применение стратегии проверки ложности недоказанных условий корректности к такому УК аналогично описанному ранее случаю, однако наличие семантических меток в определении функции *rep* позволило сгенерировать более подробный отчет о локализации ошибок:

```
This formula corresponds to lines 1-9 in function "grt_eql_key".
Its purpose is to show unsatisfiable case. Hence, given
    - assumption that precondition from line 1 holds,
    - assumption that postcondition hypothesis from line 2 holds,
ensure that
    - postcondition goal from line 2
      with substitution loop effect from lines 7-8 by rep
        that corresponds to the following loop body
        sequence of the following statements
        from line 8 to line 8
          if statement from line 8
```

```
        with condition "a[i] < key" from line 8
        and with the following positive branch
        sequence of the following statements
        from line 8 to line 8
          assignment statement "result = 1" from line 8
          break statement from line 8
    does not hold
  - break statement execution in the loop body from line 8
    holds
  - exists such i (0 <= i < n)
    that condition
    a[i] < key
    holds
```

В отличие от объяснения, полученного в результате первого эксперимента, данное объяснение содержит текст, соответствующий телу цикла, и текст, описывающий содержащее ошибку условие. Данный текст позволяет обратить внимание пользователя системы верификации на содержащее ошибку условие инструкции `if`, в ветви которой находится `break`.

Теперь рассмотрим эксперимент по автоматической дедуктивной верификации функции *grt-eql-cnt* [126].

Было сгенерировано следующее УК:

$$\forall n, key, a$$
$$((0 < n \wedge n \leq len(a) \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge grt\text{-}eql\text{-}cnt(n, key, a) = 0 \rightarrow$$
$$rep(n, key, a, 0).result = 0) \wedge$$
$$(0 < n \wedge n \leq len(a) \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge grt\text{-}eql\text{-}cnt(n, key, a) > 0 \rightarrow$$
$$rep(n, key, a, 0).result = 1))$$

Применение стратегии для программ с выходом из цикла, описанной в разделе 2.5.2, привело к генерации леммы $vc\text{-}1\text{-}lemma\text{-}1$:

$$\forall n, key, a$$
$$(0 < n \wedge n \leq len(a) \wedge n \in Int \wedge key \in Int \wedge a \in IntArr \wedge grt\text{-}eql\text{-}cnt(n, key, a) = 0 \rightarrow$$
$$\neg rep(n, key, a, 0).loop\text{-}break)$$

и к генерации леммы $vc\text{-}1\text{-}lemma\text{-}2$:

$$\forall n, key, a$$
$$(0 < n \land n \le len(a) \land n \in Int \land key \in Int \land a \in IntArr \land grt\text{-}eql\text{-}cnt(n, key, a) > 0 \to rep(n, key, a, 0).loop\text{-}break)$$

Данные леммы о свойствах финитных итераций были автоматически доказаны системой ACL2 и добавлены в теорию предметной области.

Система ACL2 автоматически доказала условие корректности, используя индукцию по n и данные леммы.

4.4. Программы с финитными итерациями над изменяемыми массивами с инструкциями `break`

Рассмотрим эксперименты по применению комплексного подхода к C-программам с финитными итерациями, которые относятся к классу итераций над изменяемыми массивами с инструкциями `break` [117, 118, 121].

4.4.1. Изменение знака первого отрицательного элемента массива на противоположный

Рассмотрим программу `negate_first` из соревнования по верификации [100]:

```
void negate_first(int n, int* a) {
    int i;
    for (i = 0; i < n; i++) {
        if (a[i] < 0) {a[i] = -a[i]; break;}}}
```

Эта программа изменяет знак первого отрицательного элемента массива на противоположный [117].

Данная программа содержит цикл с инструкцией `break`, но методы комплексного подхода позволяют работать с такими финитными итерациями.

Приведем предусловие:

$$(a_0 = a) \land (0 < n) \land (n \le length(a_0))$$

Приведем постусловие:

$$(\neg found_negative(n,\ a_0) \rightarrow a = a_0) \wedge$$
$$(found_negative(n,\ a_0) \rightarrow a = update(a_0, count_index(n,\ a_0), -a_0[count_index(n,\ a_0)]))$$

В постусловии используется предикат $found$-$negative$, заданный в теории предметной области. Этот предикат проверяет наличие отрицательного элемента в массиве. Также в постусловии используется функция $count$-$index$, заданная в теории предметной области. Эта функция вычисляет индекс первого отрицательного элемента массива в случае наличия в массиве такого элемента. Значение этой функции не определено в других случаях.

Сначала рассмотрим первый эксперимент по автоматизированной локализации ошибки в функции `negate_first` [126]. Рассмотрим функцию с внесенной ошибкой:

```
1. /*@ requires (a0 = a) && (0 < n) && (n <= length(a0));
2.     ensures (!found_negative(n, a0) ==> a == a0) &&
3.             (found_negative(n, a0) ==> a == update(a0,
                         count_index(n, a0), -a0[count_index(n, a0)]))
4. */
5. void negate_first(int n, int* a) {
6.   int i;
7.   for (i = 0; i < n; i++) {
8.        if (a[i] < a[i]) {a[i] = -a[i]; break;}}}
```

Ошибка состоит в использовании `a[i]` вместо `0` в условии `if` на строке 8.

Для данной программы с помощью правила вывода, описанного в разделе 3.2.3, было сгенерировано следующее УК (формула vc-1):

$$\forall n, a_0, a$$
$$((\lceil a_0 = a \wedge 0 < n \wedge n \leq length(a_0)\rceil^{ass_pre(1)} \wedge n \in Int \wedge$$
$$a_0 \in IntArr \wedge a \in IntArr \wedge \neg\lceil found_negative(n,\ a_0)\rceil^{ass_post(2)} \rightarrow$$
$$\lceil\lceil rep(n,\ a).a\rceil^{rep_iter(7-8)} = a_0\rceil^{ens_post(2)}) \wedge$$
$$(\lceil a_0 = a \wedge 0 < n \wedge n \leq lenght(a_0)\rceil^{ass_pre(1)} \wedge n \in Int \wedge$$
$$a_0 \in IntArr \wedge a \in IntArr \wedge \lceil found_negative(n,\ a_0)\rceil^{ass_post(3)} \rightarrow$$
$$\lceil\lceil rep(n,\ a).a\rceil^{rep_iter(7-8)} =$$
$$update(a_0, count_index(n, a_0), -a0[count_index(n, a_0)])\rceil^{ens_post(3)}))$$

Отметим, что это УК снабжено семантическими метками. Применение стратегий не позволило доказать данное УК и для него было сгенерировано следующее объяснение:

```
This formula corresponds to lines 1-8
in function "negate_first".
Hence, given
    - assumption that precondition
      from line 1 holds,
    - assumption that postcondition hypothesis
      from line 3 holds,
ensure that postcondition goal from line 3
    with substitution loop effect
    from lines 7-8 by rep
```

Рассмотрим применение стратегии поиска циклов с неиспользуемыми присваиваниями элементам массива, описанной в разделе 3.1.2. Попытаемся доказать свойство, которое проверяет, что в теле цикла есть присваивание элементами массива и элементы массива после исполнения цикла равны элементам массива до исполнения цикла. В результате применения такой стратегии генерируется следующая лемма:

$$\forall n, a_0, a$$
$$((a_0 = a) \wedge (0 < n) \wedge (n \leq length(a_0))) \rightarrow (a = rep(n, a).a)$$

Эта лемма означает, что присваивание в теле цикла может быть никогда не используемым. Так как эта лемма была доказана системой ACL2, то в результате применения данной стратегии было сгенерировано следующее предупреждение:

```
- the following warning about loop
    assumption that precondition
    from line 1 holds,
    ensures that
    array update "a[i] = -a[i]"
    from line 8
```

```
   is always unused
  has been generated
```

Объяснение УК и данное предупреждение позволяют пользователю системы верификации обратить внимание на условие инструкции `if` в теле цикла.

Теперь рассмотрим эксперимент по автоматической дедуктивной верификации функции `negate_first` [117].

Было сгенерировано следующее УК:

$$\begin{gathered}\forall n, a_0, a\\((a_0 = a) \wedge (0 < n) \wedge (n \leq length(a_0))) \rightarrow\\((\neg found_negative(n,\ a_0) \rightarrow rep(n,\ a).a = a_0) \wedge\\(found_negative(n,\ a_0) \rightarrow rep(n, a).a =\\update(a_0, count_index(n,\ a_0), -a_0[count_index(n,\ a_0)])))\end{gathered}$$

Дополнительные материалы, включая определение функции rep на языке системы ACL2, приведены в репозитории [115].

Отметим, что постусловие программы описывает два случая: исполнилась ли инструкция `break` или нет. Применение стратегии для программ, постусловием которых является разбор случаев выхода из цикла, описанной в разделе 2.4.4, привело к генерации следующей леммы:

$$\begin{gathered}((a_0 = a) \wedge (0 < n) \wedge (n \leq length(a_0))) \rightarrow\\(rep(n,\ a).loop\text{-}break = found_negative(n,\ a_0) \wedge\\(\neg found_negative(n,\ a_0) \rightarrow rep(n,\ a).a = a_0) \wedge\\(found_negative(n,\ a_0) \rightarrow rep(n, a).a =\\update(a_0, count_index(n,\ a_0), -a_0[count_index(n,\ a_0)])))\end{gathered}$$

Данная лемма была автоматически доказана системой ACL2 и добавлена в теорию предметной области. Система доказательства ACL2 успешно доказывает УК, используя индукцию по n и эту лемму.

4.4.2. Сортировка простыми вставками с инвариантом внешнего цикла и без инварианта внутреннего цикла

Рассмотрим эксперимент по автоматической верификации в модифицированной системе C-lightVer программы сортировки простыми вставками с инва-

риантом внешнего цикла и без инварианта внутреннего цикла [118]:

```
/* P */
void insertion_sort(int a[], int n){
        int k, i, j;
        /* INV */
        for (i = 1; i < n; i++){
                k = a[i];
                for (j = i - 1; j >= 0; j--){
                        if (a[j] <= k) break;
                        a[j + 1] = a[j];}
                    a[j + 1] = k;}}
/* Q */
```

Отметим, что часто инвариант внутреннего цикла имеет более сложную структуру, чем инвариант внешнего цикла, как в случае рассматриваемой программы сортировки [32]. Поэтому, отсутствие необходимости задавать инвариант внутреннего цикла является упрощением задачи для пользователя системы верификации.

Предусловие программы, инвариант и постусловие имеют следующий вид:

$$P \equiv 0 < n \wedge a = a_0 \wedge n \leq len(a_0),$$
$$INV \equiv i \leq n \wedge n \leq len(a) \wedge len(a_0) = len(a) \wedge$$
$$a_0[i : n-1] = a[i : n-1] \wedge perm(0, i-1, a_0, a) \wedge ord(0, i-1, a),$$
$$Q \equiv perm(0, n-1, a_0, a) \wedge ord(0, n-1, a),$$

где $perm(i, j, a_0, a)$ обозначает, что массив a является перестановкой массива a_0 от i-го до j-го элемента, $ord(i, j, a)$ обозначает, что массив a упорядочен от i-го до j-го элемента. Ограничения на длину возникают из-за моделирования C-массивов списками языка системы ACL2.

Применение правила вывода для цикла из аксиоматической семантики для языка C-kernel приводит к появлению трех условий корректности: условие входа в цикл, условие выхода из цикла и условие продолжения итерации.

Рассмотрим эти условия корректности. Пусть $R(u_1 \leftarrow v_1, ..., u_m \leftarrow v_m)$ обозначает одновременные подстановки выражений v_i в R вместо u_i $(1 \leq i \leq m)$.

Тогда условие входа в цикл имеет вид $P \to INV(i \leftarrow 1)$, где 1 – начальное значение счетчика итераций. В данном эксперименте будем записывать все формулы в обычной инфиксной нотации. Условия корректности и леммы на языке системы ACL2 приведены репозитории [116]. УК было автоматически сгенерировано, названо vc-1, размещено в файле vc-1.$lisp$ и доказано системой ACL2 в автоматическом режиме.

Условие выхода из цикла имеет вид $i \geq n \wedge INV \to Q$, где $i \geq n$ – отрицание условия цикла. Оно было автоматически сгенерировано, названо vc-2, размещено в файле vc-2.$lisp$ и доказано системой ACL2 в автоматическом режиме с использованием библиотеки $arithmetic$-5 (мы всегда используем эту библиотеку как теорию для арифметики).

Условие продолжения итерации имеет следующий вид:

$$\begin{gathered} i < n \wedge INV \to \\ (((((INV(i \leftarrow i+1))(a \leftarrow update\text{-}nth(j+1,k,a)))(j \leftarrow j+1)) \\ (j \leftarrow rep((i-1)+1, envir\text{-}init(i-1,k), frame\text{-}init((i-1)+1,a)).j, \\ a \leftarrow rep((i-1)+1, envir\text{-}init(i-1,k), frame\text{-}init((i-1)+1,a)).a)) \\ (k \leftarrow a[i])), \end{gathered}$$

где $i < n$ – условие цикла. Применение символического метода верификации финитных итераций позволило избежать использования инварианта внутреннего цикла. Оно было автоматически сгененрировано, названо vc-3, размещено в файле vc-3.$lisp$. Отметим, что это условие корректности основано на использовании операции замены.

Рассмотрим доказательство условия продолжения итерации. Данное условие корректности не удалось доказать в системе ACL2 в автоматическом режиме. Поэтому, при его доказательстве были применены стратегии доказательства УК. Рассмотрим ключевые леммы, которые были добавлены в предметную область в результате работы стратегий. Эти леммы позволили системе ACL2 доказать условие корректности.

Установим соответствие между обозначениями стратегий, описанными в разделах 2.3.1, 2.3.2, 2.5.1, 2.5.3, и вложенной финитной итерацией вместе с условием ее продолжения. В нашем случае параметр a обозначает массив a,

параметр n обозначает переменную i, параметр i обозначает переменную j.

Таким образом, в условии корректности вызов функции *rep* имеет следующие аргументы:

- при вызове функции *rep* в условии корректности первым аргументом является количество итераций, т.е. $(i-1)+1$;
- в нашем случае значением *env.upper-bound* является $(i-1)+1$. Также в нашем случае структура *envir* имеет поле k. Значением $env.k$ является $a[i]$. При вызове функции *rep* в условии корректности в качестве аргумента используется выражение $envir\text{-}init((i-1)+1, a[i])$. Значением этого выражения является описанный объект *env*;
- в нашем случае значением $fr.j$ является $i-1$. В нашем случае значением $fr.loop\text{-}break$ является *nil*. В нашем случае значением $fr.a$ является a. При вызове функции *rep* в условии корректности в качестве аргумента используется выражение $frame\text{-}init((i-1)+1, a)$. Значением этого выражения является описанный объект fr.

Значит,

- значением выражения $rep((i-1)+1, env, fr).j$ является значение счетчика цикла, полученное в результате итерации;
- значением выражения $rep((i-1)+1, env, fr).loop\text{-}break$ является индикатор, сработал ли оператор `break`;
- значением выражения $rep((i-1)+1, env, fr).a$ является массив, полученный в результате итерации.

Параметр T обозначает посылку УК, то есть $INV \wedge (i \leq n)$.

Пусть e обозначает массив, полученный в результате цикла, то есть массив $update\text{-}nth(rep((i-1)+1, env, fr).j+1, a[i], rep((i-1)+1, env, fr).a)$.

Сначала была использована стратегия для финитных итераций над изменяемыми массивами, описанная в разделе 2.5.3. В результате анализа цикла

было установлено, что в левой части оператора присваивания стоит элемент массива с индексом `j + 1`. При трансляции данного выражения на язык ACL2 получаем выражение $(env.upper\text{-}bound - iteration) + 1$, согласно правилам генерации функции *rep*. Следовательно, по стратегии для финитных итераций над изменяемыми массивами была сгенерирована следующая формула:

$$\begin{array}{c} (iteration \in N) \wedge (index \in N) \wedge \\ (iteration \leq env.upper\text{-}bound) \wedge (env.upper\text{-}bound < (len fr.a)) \wedge \\ (fr.j = (env.upper\text{-}bound - 1)) \wedge (index \neq (env.upper\text{-}bound - iteration) + 1) \rightarrow \\ rep.a(iteration - 1, env, fr)[index] = rep.a(iteration, env, fr)[index] \end{array},$$

Данная лемма была автоматически доказана системой ACL2, ее можно найти в файле *rep.lisp* под названием `rep-lemma-22`. С ее помощью успешно доказывается следующая цель из условия корректности: $a0[i + 1 : n - 1] = e[i + 1 : n - 1]$.

Далее была применена стратегия для программ, спецификации которых содержат функции со свойством конкатенации, описанная в разделе 2.3.2. Теоремы *permutation*-7 и *ordered*-3 в файлах *permut* и *ordered* соответственно позволили доказать, что предикат *perm* удовлетворяет свойству конкатенации, а предикат *ordered*-3 удовлетворяет свойству конкатенации со склейкой на границе по отношению $\leq$.

Следовательно, рассмотрим леммы, возникающие при доказательстве двух целей. Обозначим как A цель, содержащую предикат *perm*. Обозначим как B цель, содержащую предикат *ord*. Тогда обозначим как G лемму $(INV \wedge (i \leq n)) \rightarrow A$ и обозначим как H лемму $(INV \wedge (i \leq n)) \rightarrow B$.

Рассмотрим сначала применение шагов стратегии к формуле G.

Вначале проверяется истинность утверждения об эквивалентности

$$a[0 : rep((i - 1) + 1, env, fr).j] = (rep((i - 1) + 1, env, fr).a)[0 : rep((i - 1) + 1, env, fr).j].$$

С помощью стратегии выбора посылок, описанной в разделе 2.3.1, это утверждение преобразуется в следующую формулу:

$$\begin{array}{c} (iteration \in N) \wedge (iteration \leq env.upper\text{-}bound) \wedge \\ (env.upper\text{-}bound < (len fr.a)) \wedge (fr.j = (env.upper\text{-}bound - 1)) \rightarrow \\ a[0 : rep(iteration, env, fr).j] = (rep(iteration, env, fr).a)[0 : rep(iteration, env, fr).j] \end{array},$$

Данная лемма была сгенерирована в файле *rep.lisp*, названа `rep-lemma-76` и доказана с помощью леммы `rep-lemma-22`. После успешного доказательства в соответствии со стратегией выбора посылок была сгенерирована новая формула:

$$\begin{gathered}(INV \wedge (i \leq n)) \rightarrow \\ a[0 : rep((i-1)+1, env, fr).j] = (rep((i-1)+1, env, fr).a)[0 : rep((i-1)+1, env, fr).j]\end{gathered},$$

Данная лемма была сгенерирована в файле *vc-3.lisp* как `vc-3-lemma-18` и автоматически доказана с помощью леммы `rep-lemma-76`. Посылка условия продолжения итерации и лемма `vc-3-lemma-18` позволили автоматически доказать перестановочность массива a и массива $rep((i-1)+1, env, fr).a$ в диапазоне $[0 : rep((i-1)+1, env, fr).j]$. Это позволило автоматически доказать перестановочность массива a и массива e в диапазоне $[0 : rep((i-1)+1, env, fr).j]$

При анализе цикла была обнаружена операция `a[j + 1] = a[j]`, то есть возможный сдвиг массива на 1. Было проверено на истинность утверждение

$$a[rep((i-1)+1, env, fr).j+1 : i-1] = (rep((i-1)+1, env, fr).a)[rep((i-1)+1, env, fr).j+2 : i]$$

С помощью стратегии выбора посылок это утверждение преобразуется в следующую формулу:

$$\begin{gathered}(iteration \in N) \wedge (iteration \leq env.upper\text{-}bound) \wedge \\ (env.upper\text{-}bound < (lenfr.a)) \wedge (fr.j = (env.upper\text{-}bound - 1)) \rightarrow \\ a[rep(iteration, env, fr).j+1 : i-1] = (rep(iteration, env, fr).a)[rep(iteration, env, fr).j+2 : i]\end{gathered}$$

Данная лемма была сгенерирована в файле *rep.lisp*, названа `rep-lemma-55` и автоматически доказана с помощью леммы `rep-lemma-22`. После успешного доказательства в соответствии со стратегией выбора посылок была сгенерирована новая формула:

$$\begin{gathered}(INV \wedge (i \leq n)) \rightarrow \\ a[rep((i-1)+1, env, fr).j+1 : i-1] = (rep((i-1)+1, env, fr).a)[rep((i-1)+1, env, fr).j+2 : i]\end{gathered}$$

Данная лемма была сгенерирована в файле *vc-3.lisp* как `vc-3-lemma-27` и автоматически доказана с помощью леммы `rep-lemma-55`.

Посылка условия продолжения итерации и лемма `vc-3-lemma-27` позволили доказать перестановочность массива $a[rep((i-1)+1, env, fr).j+1 : i-1]$ и массива $(rep((i-1)+1, env, fr).a)[rep((i-1)+1, env, fr).j+2 : i]$. Это позволило автоматически доказать перестановочность массива $a[rep((i-1)+1, env, fr).j+1 : i-1]$ и массива $e[rep((i-1)+1, env, fr).j+2 : i]$.

Автоматически доказывается, что $e[rep((i-1)+1, env, fr).j+1 : rep((i-1)+1, env, fr).j+1] = a[i : i]$. Это позволяет автоматически доказать перестановочность массива $a[i : i]$ и массива $e[rep((i-1)+1, env, fr).j+1 : rep((i-1)+1, env, fr).j+1]$. Так как выполняется перестановочность массива $a[rep((i-1)+1, env, fr).j+1 : i-1]$ и массива $e[rep((i-1)+1, env, fr).j+2 : i]$ и перестановочность массива $a[i : i]$ и массива $e[rep((i-1)+1, env, fr).j+1 : rep((i-1)+1, env, fr).j+1]$, то свойство конкатенации перестановочности позволяет автоматически доказать перестановочность массивов a и e в диапазоне [rep((i - 1) + 1, env, fr).j + 1 : i]. Так как ранее была доказана перестановочность массивов a и e в диапазоне [0 : rep((i - 1) + 1, env, fr).j], то свойство конкатенации позволяет автоматически доказать перестановочность массива a и массива e в диапазоне [0 : i].

Данные леммы и посылка условия корректности позволили системе ACL2 автоматически доказать перестановочность массива $a0$ и массива e в диапазоне [0 : i], используя свойство конкатенации.

Рассмотрим теперь применение шагов стратегия для программ, спецификации которых содержат функции со свойством конкатенации, к формуле H.

Леммы `vc-3-lemma-27` и `rep-lemma-55` об эквивалентности подмассивов уже были доказаны. Так как для предиката ord выполняется свойство конкатенации со склейкой на границах, то необходимо проверить выполнение отношения $\leq$ на границах. В ходе анализа цикла было установлено, что он содержит инструкцию `break`. Была применена стратегия для финитных итераций с инструкцией `break`, описанная в разделе 2.5.1. Следующее выражение является `break`-условием: $(a[rep(iteration-1, env, fr).j]) <= env.k$.

С помощью стратегии выбора посылок утверждения, предлагаемые для проверки стратегией для финитных итераций с инструкцией `break`, преобразу-

ется в следующие формулы:

$$(iteration \in N) \wedge (iteration \leq env.upper\text{-}bound) \wedge$$
$$(env.upper\text{-}bound < (len fr.a)) \wedge (fr.j = (env.upper\text{-}bound - 1)) \wedge$$
$$(rep(iteration, env, fr).j + 2) \leq env.upper\text{-}bound \wedge rep(iteration, env, fr).loop\text{-}break \rightarrow$$
$$rep(iteration - 1, env, fr).j = rep(iteration, env, fr).j$$

$$(iteration \in N) \wedge (iteration \leq env.upper\text{-}bound) \wedge$$
$$(env.upper\text{-}bound < (len fr.a)) \wedge (fr.j = (env.upper\text{-}bound - 1)) \wedge$$
$$(rep(iteration, env, fr).j + 2) \leq env.upper\text{-}bound \wedge \neg rep(iteration, env, fr).loop\text{-}break \rightarrow$$
$$rep(iteration, env, fr).j = env.upper\text{-}bound - iteration - 1$$

Эти леммы были сгенерированы в файле *rep.lisp*, названы `rep-lemma-7` и `rep-lemma-8`, и доказаны автоматически. После успешного доказательства можно перейти к утверждению о `break`-условии.

В рассматриваемом случае утверждение о `break`-условии, предлагаемое для проверки стратегией для финитных итераций с инструкцией `break`, имеет следующий вид:

$$a[rep((env.upper\text{-}bound - rep(env.upper\text{-}bound, env, fr).j) - 1, env, fr).j] > env.k$$

$$a[rep(env.upper\text{-}bound - rep(env.upper\text{-}bound, env, fr).j, env, fr)] <= enk.k$$

В итоге, леммы, соответствующие данным утверждениям, были сгенерированы в файле *vc-3.lisp*, названы `vc-3-lemma-78` и `vc-3-lemma-79`, и были автоматически доказаны.

Данные леммы и посылка условия корректности, а также леммы полученные для формулы G, позволили системе ACL2 автоматически доказать упорядоченность массива e в диапазоне [0 : i]. Итого, все условие корректности было автоматически доказано с помощью набора стратегий комплексного подхода.

4.4.3. Сортировка простыми вставками без инвариантов циклов

Опишем эксперимент по автоматизированной локализации ошибки в сортировке простыми вставками и эксперимент по автоматической верификации данной сортировки [121].

Рассмотрим программу на языке C-light без инвариантов циклов, реализующую сортировку простыми вставками:

```
 1. /* P */
 2. void insertion_sort(int a[], int n){
 3.    int k, i, j;
 4.    for (i = 1; i < n; i++){
 5.       k = a[i];
 6.       for (j = i - 1; j >= 0; j--){
 7.          if (a[j] <= k) break;
 8.          a[j + 1] = a[j];}
 9.       a[j + 1] = k;}}
10. /* Q */
```

где спецификации записаны в конструкциях, задающих на языке C комментарии. P и Q обозначают предусловие и постусловие соответственно.

Отметим, что данная программа не содержит явных операций взятия адреса и разыменования указателей. Поэтому, метод смешанной аксиоматической семантики [147] позволяет нам использовать в данном случае простую модель памяти. Это приводит к упрощению УК.

Также отметим, что данная программа относится к классу программ с вложенными циклами.

Спецификации и формулы в данном эксперименте были заданы на языке системы ACL2, но для записи формул при описании данного эксперимента будем используем классическую инфиксную нотацию.

Предусловие P является следующей формулой: $0 < n \leq length(a_0) \land a = a_0$, где

- $length$ — функция, принимающая на вход массив и вычисляющая его длину.

- a_0 – вспомогательный массив, который хранит исходное значение массива a.

Постусловие Q является следующей формулой:

$$perm(0, n-1, a_0, a) \wedge ord(0, n-1, a),$$

где

- $perm$ — предикат, задающий свойство перестановочности.

- ord — предикат, задающий свойство упорядоченности.

Конъюнкция свойств перестановочности и упорядоченности является свойством отсортированности массива.

Первым аргументом предиката $perm$ является нижняя граница диапазона индексов. Вторым аргументом $perm$ является верхняя граница диапазона индексов. Третьим и четвертым аргументами предиката $perm$ являются массивы. Предикат $perm$ проверяет, являются ли эти массивы перестановкой друг друга в диапазоне от нижней границы (включая нижнюю границу) до верхней границы (включая верхнюю границу).

Первым аргументом предиката ord является нижняя граница диапазона индексов. Вторым аргументом предиката ord является верхняя граница диапазона индексов. Третьим аргументом предиката ord является массив. Предикат ord проверяет, является ли этот массив упорядоченным по возрастанию в диапазоне от нижней границы (включая нижнюю границу) до верхней границы (включая верхнюю границу).

Предикаты $perm$ и ord являются предикатами предметной области. Мы задали модуль для системы ACL2 с определением предиката $perm$ и с теоремами о предикате $perm$. Этот модуль задан в файле "permut.lisp". Данный файл доступен в репозитории [113]. Также мы задали модуль для системы ACL2 с

определением предиката ord и с теоремами о предикате ord. Этот модуль задан в файле "ordered.lisp". Данный файл также доступен в репозитории [113].

Сначала рассмотрим эксперимент по автоматизированной локализации ошибки [121]. Рассмотрим программу на языке C-light без инвариантов циклов, реализующую сортировку простыми вставками с внесенной ошибкой:

```
 1. /* P */
 2. void insertion_sort(int a[], int n){
 3.    int k, i, j;
 4.    for (i = 1; i < n; i++){
 5.       k = a[i];
 6.       for(j=i;j>=0;j--){
 7.          if (a[j] <= k) break;
 8.          a[j + 1] = a[j];}
 9.       a[j + 1] = k;}}
10. /* Q */
```

Ошибка состоит в том, что инициализирующей инструкцией вложенного цикла `for` является инструкция `j=i` вместо инструкции `j=i-1`.

Было сгенерировано следующее УК:

$$0 < n \leq length(a_0) \wedge a = a_0 \rightarrow$$
$$perm(0, n-1, a_0, rep_1(n, a, 1).a) \wedge ord(0, n-1, rep_1(n, a, 1).a),$$

где

- rep_1 — операция замены для внешнего цикла.
- 1 — начальное значение счетчика внешнего цикла.
- $rep_1(n, a, 1).a$ — массив a после исполнения внешнего цикла.

Определения функций rep_1 и rep_2 снабжены семантическими метками в результате применения алгоритма генерации операции замены, описанного в разделе 3.2.4. Определение функции rep_1 основано на применении операции замены rep_2 для внутреннего цикла. Ошибка в программе приводит к тому, что функция rep_2 применяется в теле функции rep_1 с неправильными аргументами.

Вместо начального значения счетчика цикла $i-1$ в качестве аргумента функции rep_2 используется i.

Рассмотрим влияние ошибки в программе на исполнение i-й итерации внешнего цикла. Ошибка приводит к тому, что внутренний цикл всегда завершает исполнение на первой итерации из-за истинности условия инструкции if, содержащей `break` в положительной ветви. После завершения внутреннего цикла происходит присваивание элементу массива с индексом $i+1$ элемента массива с индексом i. В итоге, данное УК является истинным, когда элемент массива с индексом 0 имеет произвольное значение, а все остальные элементы совпадают и их значение не меньше значения нулевого элемента. Элемент с индексом 0 может отличаться от всех остальных, так как начальным значением счетчика внешнего цикла является 1, а не 0. Во всех остальных случаях это УК является ложным.

Система ACL2 не доказала истинность данного УК. Также система ACL2 не доказала ложность этого УК во всех случаях, используя проверку на истинность отрицания данного УК. Эта стратегия локализации ошибок не приводит к успеху из-за истинности УК в некоторых случаях. Модуль локализации ошибок в модифицированной системе C-lightVer последовательно применяет стратегию поиска циклов с неиспользуемыми присваиваниями элементам массива, описанную в разделе 3.1.2, и стратегию проверки исполнения инструкции `break` на первой итерации цикла, описанную в разделе 3.1.3.

Ошибка в программе приводит к тому, что вложенный цикл не изменяет массив a. Отметим, что вложенный цикл содержит присваивание элементам массива. Применение стратегии поиска циклов с неиспользуемыми присваиваниями элементам массива приводит к успешному доказательству соответствующей леммы. В репозитории [113] в файле "warnings.lisp" эта лемма задана как `warnings-lemma-11`.

Также ошибка в программе приводит к тому, что исполнение вложенного цикла всегда завершается на первой итерации. Инструкция `break` вложенного цикла всегда исполняется на первой итерации и вложенный цикл не изменяет значение счетчика цикла (переменная j). Поэтому, применение стратегии

проверки исполнения инструкции `break` на первой итерации цикла приводит к успешному доказательству соответствующей леммы. В репозитории [113] в файле "warnings.lisp" эта лемма задана как `warnings-lemma-9`. Условие `a[i] <= a[i]` было символически вычислено, используя подстановку i вместо j и постановку $a[i]$ вместо k в условие инструкции if. Это условие исполнения инструкции `break` во вложенном цикле. Это условие вычислено для отчета о локализации ошибки. Отметим, что данное условие всегда истинно и является условием инструкции `if`. Эта информация является еще одним признаком наличия ошибки в программе.

В итоге, алгоритм генерации объяснений недоказанных УК для программ с вложенными циклами генерирует отчет, состоящий из трех частей. Рассмотрим первую часть отчета, которая является результатом применения метода локализации ошибок:

```
This formula corresponds to lines 1-10 in
function "insertion_sort". This formula has
not been proven by C-ligthVer system using
ACL2 prover. Hence,
- the following explanation of this formula
    precondition from line 1
        implies
    postcondition goal from line 10
      with substitution loop effect
      from lines 4-9 by rep1
        that corresponds to the loop with
        initialization expression "i = 1" from line 4
        condition expression "i < n" from line 4
        iteration expression "i++" from line 4
        and the following loop body
        sequence of the following statements from line 5 to line 9
          assignment statement "k = a[i]" from line 5
          substitution of inner loop effect from lines 6-8 by rep2
          that corresponds to the inner loop
          with
            initialization expression "j=i" from line 6
```

```
            condition expression "j>=0" from line 6
            iteration expression "j--" from line 6
            and the following loop body
            sequence of
            the following statements from line 7 to line 8
              if statement from line 7
                with condition "a[j] <= k" from line 7
                and with
                the following positive branch
                sequence of
                the following statements from line 7 to line 7
                  break statement from line 7
              array update "a[j + 1] = a[j]" from line 8
          array update "a[j + 1] = k" from line 9
  has been generated
```

Эта часть отчета позволяет сопоставить подформулы УК и исходную программу, а также сопоставить определение операции замены для внешнего цикла и исходную программу. Снабжение семантическими метками определения операции замены для внутренних циклов позволило сопоставить определение операции замены для внутреннего цикла и исходную программу.

Рассмотрим вторую часть отчета, которая является предупреждением о возможной ошибке, генерируемым стратегией поиска циклов с неиспользуемыми присваиваниями элементам массива:

```
- the following warning about inner loop
    assumption that precondition
    from line 1 holds,
    ensures that
    array update "a[j + 1] = a[j]"
    from line 8
    is always unused
  has been generated
```

Эта часть отчета позволяет обратить внимание пользователя на вложенный цикл.

Рассмотрим третью часть отчета, которая состоит из двух предупреждений о возможной ошибке, генерируемых стратегией проверки исполнения инструкции `break` на первой итерации цикла:

```
- the following warning about inner loop
    assumption that precondition
    from line 1 holds,
    ensures that
    break statement
    from line 7 always executes
    at first iteration
  has been generated
- the following warning about inner loop
    assumption that precondition
    from line 1 holds,
    ensures that
    break condition "a[j] <= k"
    from line 7
    is always true at first iteration as
    "a[i] <= a[i]"
  has been generated
```

Эта часть отчета также позволяет пользователю обратить внимание на вложенный цикл. Кроме того, эта часть отчета позволяет обратить внимание пользователя на условие исполнения `break`.

Данный отчет позволяет обратить внимание пользователя на вложенный цикл и локализовать ошибку. Отметим, что пользователю не нужно просматривать всю программу, чтобы локализовать ошибку. Вместо изучения сложной структуры УК пользователю предоставляется текст о сопоставлении подформул УК и исходной программы. Комплексный подход к автоматизации дедуктивной верификации C-программ, реализованный в модифицированной системе C-lightVer, позволяет упростить и автоматизировать локализацию ошибки в данном эксперименте.

Теперь рассмотрим эксперимент по автоматической дедуктивной верификации программы сортировки простыми вставками без использования инвари-

антов циклов [121]. Дополнительные материалы по данному эксперименту приведены в репозитории [113].

Было сгенерировано следующее УК:

$$0 < n \leq length(a_0) \land a = a_0 \rightarrow$$
$$perm(0, n-1, a_0, rep_1(n, a, 1).a) \land\ ord(0, n-1, rep_1(n, a, 1).a),$$

где

- rep_1 — операция замены для внешнего цикла.
- 1 — начальное значение счетчика внешнего цикла.
- $rep_1(n, a, 1).a$ — массив a после исполнения внешнего цикла.

В репозитории [113] в файле "vc-1.lisp" данное УК задано как `vc-1-lemma-103`. Определение rep_1 задано в репозитории [113] в файле "rep1.lisp" как `rep1`. Определение rep_2 задано в репозитории [113] в файле "rep2.lisp" как `rep2`. Определение функции rep_1 основано на определении функции rep_2. Следовательно, УК содержит неявную композицию функций rep_1 и rep_2.

Система ACL2 не доказала истинность данного УК без применения стратегий доказательства. Так как спецификации УК содержат предикаты, которые могут удовлетворять свойству конкатенации, то модуль управления доказательством применил стратегию для программ, спецификации которых содержат функции со свойством конкатенации, описанную в разделе 2.3.2. Так как данное УК содержит вложенные циклы, то модуль управления доказательством применил стратегию для программ с финитными итерациями над массивами, описанную в разделе 2.3.3, и стратегию для программ с вложенными циклами, описанную в разделе 2.5.4.

Спецификации рассматриваемой программы содержат функции со свойством конкатенации, так как для предиката $perm$ выполнено свойство конкатенации и для предиката ord выполнено свойство конкатенации со склейкой на границе.

Предикат $perm$ удовлетворяет свойству конкатенации, так как в теории предметной области содержится следующая лемма о свойствах $perm$:

$$(i \in N \wedge j \in N \wedge k \in N \wedge i \leq k \wedge k < j \wedge j < length(u) \wedge j < length(v) \wedge perm(i, k, u, v) \wedge perm(k+1, j, u, v)) \rightarrow perm(i, j, u, v),$$

где N — множество натуральных чисел. Выполнение свойства конкатенации автоматически доказывается благодаря этой теореме. В репозитории [113] в файле "permut.lisp" эта теорема задана как `permutation-7`.

Предикат ord удовлетворяет свойству конкатенации со склейкой на границе, так как в теории предметной области содержится следующая лемма о свойствах ord:

$$(i \in N \wedge j \in N \wedge k \in N \wedge i \leq k < j \wedge j < length(u) \wedge ord(i, k, u) \wedge ord(k+1, j, u) \wedge u[k] \leq u[k+1]) \rightarrow ord(i, j, u),$$

где N — множество натуральных чисел, $u[k] \leq u[k+1]$ — условие склейки на границе для свойства конкатенации.

Эвристический анализ обнаружил в теле циклов инструкции присваивания элементам массива элементов того же самого массива. В результате, были сгенерированы леммы о равенстве следующих пар подмассивов:

1. $(rep_2(i, a, args).a)[0 : j]$ и $a[0 : j]$
2. $(rep_2(i, a, args).a)[j + 2 : i]$ и $a[j + 1 : i - 1]$,

где

- a — массив, над которым исполняется финитная итерация.
- i — количество итераций внутреннего цикла в случае, если не исполнится `break`.
- $args$ — аргументы rep_2, вычисленные с помощью обратного прослеживания.
- $rep_2(i, a, args).a$ — массив a после i-й итерации.

Для демонстрации данных равенств подмассивов рассмотрим иллюстрацию 4.2.

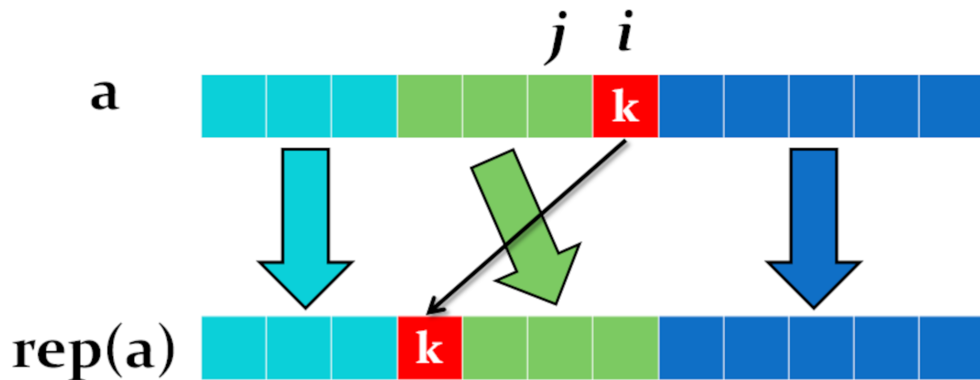

Рис. 4.2. Соотношение между частями массива до и после исполнения вложенного цикла сортировки простыми вставками

На данной иллюстрации изображено разбиение массива на части до исполнения вложенной финитной итерации и расположение данных частей после ее исполнения

Равенство подмассивов $(rep_2(i, a, args).a)[0 : j]$ и $a[0 : j]$ означает, что вложенный цикл не изменил начало массива до этой позиции j. В репозитории [113] в файле "rep2.lisp" эта теорема задана как `rep2-lemma-76`.

Равенство подмассивов $(rep_2(i, a, args).a)[j + 2 : i]$ и $a[j + 1 : i - 1]$ означает, что вложенный цикл привел к сдвигу на единицу последовательности $a[j + 1 : i - 1]$. В репозитории [113] в файле "rep2.lisp" эта теорема задана как `rep2-lemma-55`.

Данные леммы о равенстве подмассивов были автоматически доказаны и добавлены в теорию предметной области.

Сравним результаты двух смежных итераций внешнего цикла. Результат следующей итерации отличается от результата предыдущей возрастанием отсортированной части массива на один элемент и убыванием неотсортированной

части массива на один элемент.

Для демонстрации изменения обработанной части и необработанной части массива на смежных финитных итерациях рассмотрим иллюстрацию 4.3.

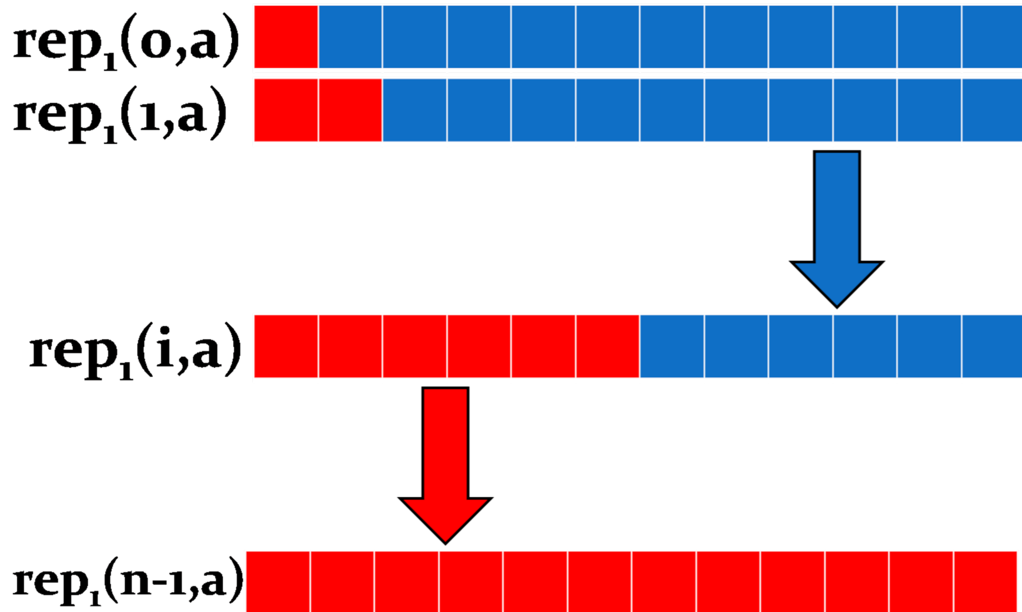

Рис. 4.3. Изменение отсортированной части и неотсортированной части массива на смежных итерациях внешнего цикла программы сортировки простыми вставками

На данной иллюстрации изображено разбиение массива на отсортированную часть и неотсортированную часть на смежных итерациях внешнего цикла.

Отметим, что на смежных итерациях необработанные части последовательностей за вставляемым элементом равны. В данном случае необработанная часть последовательности внешнего цикла — это необработанная часть последовательности внутреннего цикла. Поэтому, применение стратегии для программ с финитными итерациями над массивами приводит к успешному доказательству соответствующей леммы и добавлению этой леммы в теорию предметной области. В репозитории [113] в файле "rep2.lisp" эта лемма задана как `rep2-lemma-53`.

Рассмотрим результат i-й итерации внешнего цикла. В результате такой итерации массив делится на две части: отсортированная часть до элемента с индексом i (включающая элемент с индексом i), и неотсортированной часть (не включающая элемент с индексом i). Значение i является границей между отсортированной частью массива и неотсортированной частью массива. Поэтому,

в данном случае, стратегия автоматизации доказательства УК для программ с вложенными циклами является индукцией по границе между отсортированной и неотсортированной частью массива. Система ACL2 автоматически доказывает базу индукции.

Индукционная гипотеза является утверждением о выполнении свойств перестановочности и упорядоченности от начала массива до i-го элемента (не включая i-й элемент). Леммы, порожденные примененными стратегиями, являются утверждениями о структуре массива после исполнения i-й итерации. Система ACL2 автоматически доказывает индукционный переход, используя данные леммы. Автоматическое доказательство базы индукции и индукционного перехода приводит к автоматическому доказательству данного УК.

Комплексный подход к автоматизации дедуктивной верификации C-программ, реализованный в модифицированной системе C-lightVer, позволяет автоматически верифицировать программу сортировки простыми вставками на языке C.

4.5. Выводы

На основе сделанного описания применения комплексного подхода к дедуктивной верификации программ на языке C можно сделать следующие выводы:

- В модифицированной версии системы C-lightVer МГУК генерирует УК с применениями функций rep, выражающими результаты финитных итераций, и с семантическими метками. Это позволяет избежать задания инвариантов циклов и применить методы комплексного подхода к C-программам с финитными итерациями.

- Модуль генерации лемм в системе C-lightVer, который используется и модулем управления доказательством УК, и модулем локализации ошибок, демонстрирует, что комплексный подход позволяет единообразно реализовать проверку финитных итераций на выполнение различных свойств. К таким свойствам относятся как различные утверждения, позволяющие

доказать корректность программы, так и утверждения, истинность которых может означать наличие ошибок. Единообразие реализации состоит в том, что данные утверждения генерируются в одном модуле генерации лемм по схожим шаблонам, основанным на использовании информации из возвращаемого значения операции замены. Это позволяет упростить реализацию расширений комплексного подхода новыми стратегиями доказательства УК и локализации ошибок.

- В отличие от исходной версии системы C-lightVer, в модифицированной версии пользователю предоставляется подробный отчет на естественном языке о соответствии недоказанных УК и фрагментов программы. Такой отчет может содержать предупреждения о циклах, а также условиях выполнения `break`. Эксперименты продемонстрировали, что такой отчет может позволить пользователю найти ошибку без анализа самих недоказанных УК и промежуточной C-kernel программы.

- Исходная версия системы C-lightVer не могла доказать истинные УК, если их не доказывала система доказательства теорем с настройками по умолчанию. Комплексный подход позволил реализовать в модифицированной версии C-lightVer более универсальный способ, основанный на использовании при доказательстве УК свойств самой программы и свойств финитных итераций данной программы. Эксперименты продемонстрировали, что такой способ может позволить доказать УК, истинность которых не может проверить система доказательства теорем без этой дополнительной информации.

Глава 5

Применение разработанных методов для дедуктивной верификации Cloud Sisal программ

Опишем применение комплексного подхода не только к программам на императивном языке программирования C, но и к программам на языке Cloud Sisal [106]. Рассмотрим два реализованных в качестве модулей CPPS способа дедуктивной верификации данных программ: дедуктивная верификация промежуточного представления на языке C [123, 125] или дедуктивная верификация подмножества Cloud Sisal (язык Cloud-Sisal-kernel) [4]. Опишем разработанную аксиоматическую семантику языка Cloud-Sisal-kernel [130], позволяющую проводить дедуктивную верификацию без использования инвариантов циклических выражений. Также рассмотрим расширение языка C циклами Cloud Sisal (язык C-Sisal-kernel) [126], позволяющее применить в системе CPPS методы комплексного подхода системы C-lightVer. Продемонстрируем применение комплексного подхода при проведении экспериментов по верификации программ на языках Cloud Sisal, Cloud-Sisal-kernel и C-Sisal-kernel.

5.1. Система облачного параллельного программирования CPPS

В качестве предварительной информации рассмотрим систему облачного параллельного программирования (CPPS) [105]. Языком данной системы является Cloud Sisal [5], новейшая версия языка Sisal [39, 81, 90]. Данный язык является функциональным языком программирования [103]. Поэтому, использование языка Sisal позволяет избегать побочных эффектов [6]. Отсутствие побочных эффектов является одной из причин использования языка Sisal в системе CPPS [104]. Другой причиной является наличие в языке Sisal циклов специального вида [108]. Задача автоматического распараллеливания таких циклов решается удобным образом в некоторых случаях [108, 168].

Первый шаг работы системы CPPS состоит в трансляции Sisal-программы во внутреннее представление [104]. Второй шаг работы системы CPPS состоит в применении преобразований ко внутреннему представлению [108]. Целью таких преобразований является оптимизация времени исполнения. Главным из таких преобразований является автоматическое распараллеливание. Перед каждым преобразованием и после любого преобразования может быть сгенерирован код на целевом языке [2, 107, 168]. На текущем этапе развития системы CPPS в качестве целевых языков используются C [2] и MSIL [107, 168]. Третий шаг работы системы CPPS состоит в компиляции кода на целевом языке и исполнении полученного исполняемого файла [2, 168]. При этом, на всех этапах работы системы CPPS сохраняется эквивалентность между полученным представлением и исходной программой [108].

Система CPPS позволяет избежать явного использования конструкций параллельного программирования [4]. При этом, программа может быть эффективно распараллелена в автоматическом режиме [107]. Таким образом, язык Cloud Sisal позволяет реализовать концепцию неявного параллелизма [176]. Рассмотрим подробнее, какие конструкции языка Cloud Sisal позволяют это сделать.

5.1.1. Язык Cloud Sisal

Язык Cloud Sisal основан на специальных видах циклов [103, 105, 176]. Эти виды циклов позволяют упростить распараллеливание и векторизацию операций над массивами [105, 108, 168].

Триплетом является структура вида [*lower boundary* .. *upper boundary* .. *step*]. Она задает арифметическую прогрессию между заданными границами с фиксированным шагом. Диапазоном является структура, основанная на декартовом произведении триплетов.

Диапазоном является структура, основанная на декартовом произведении триплетов.

$$var_1 \texttt{ in } triplet_1 \texttt{ cross } var_2 \texttt{ in } triplet_2 \texttt{ cross } \ldots var_{n-1} \texttt{ in } triplet_{n-1} \texttt{ cross } triplet_n,$$

где $var_1, \ldots, var_n$ являются переменными и $triplet_1, \ldots, triplet_n$ являются триплетами.

К циклическим выражениям языка Cloud Sisal относятся следующие виды выражений:

- циклические выражения, управляемые диапазоном;
- циклические выражения с условиями завершения;
- выражения конструирования массивов;
- выражения замещения элементов массивов.

Рассмотрим циклическое выражение, управляемое диапазоном:

$$\texttt{for } var_1 \texttt{ in } triplet_1 \texttt{ cross } var_2 \texttt{ in } triplet_2 \texttt{ cross} \ldots$$
$$var_{n-1} \texttt{ in } triplet_{n-1} \texttt{ cross } var_n \texttt{ in } triplet_n \texttt{ do}$$
$$\texttt{returns } reduction\ expr \texttt{ end for}$$

где var_i и $triplet_j$ являются переменными и триплетами соответственно, *expr* является редуцируемым выражением, которое может зависеть от этих переменных, *reduction* является редукцией, декартово произведение обозначается ключевым словом `cross`. Этот цикл совершает итерации над декартовым произведением триплетов. Значением цикла после определенной итерации является значение редукции, примененное к значению *expr* на этой итерации и к значению цикла после предыдущей итерации.

Рассмотрим главные редукции:

- `array of` *expr* возвращает массив значений редуцируемого выражения *expr*;
- `value of` *expr* возвращает последнее значение редуцируемого выражения *expr*;

- `sum of` *expr* вычисляет сумму значений редуцируемого выражения *expr*;
- `product of` *expr* вычисляет произведение значений редуцируемого выражения *expr*;
- `greatest of` *expr* вычисляет наибольшее значение редуцируемого выражения *expr*;
- `least of` *expr* вычисляет наименьшее значение редуцируемого выражения *expr*.

Обозначим как *loop_e* такое циклическое выражение с такими подвыражениями.

Рассмотрим циклическое выражение с конструкцией *while*:

$$\begin{array}{l} for\ var_1\ in\ triplet_1\ cross\ \ldots\ var_n\ in\ triplet_n \\ \quad while\ condition\ do\ returns\ reduction\ expr\ end\ for \end{array}$$

где конструкция *while* соответствует следующей инструкции, записанной на языке программирования C: if (!*condition*) {**break**; }. Эта конструкция не задает вложенный цикл.

Рассмотрим выражение конструирования массива:

$$\begin{array}{c} \texttt{array}[0\ ..\ hight_1,\ 0\ ..\ hight_2,\ \ldots,\ 0\ ..\ hight_{n-1},\ 0\ ..\ hight_n]\ \texttt{of} \\ [var_1\ \texttt{in}\ triplet_1\ \texttt{cross}\ var_2\ \texttt{in}\ triplet_2\ \texttt{cross}\ \ldots \\ var_{n-1}\ \texttt{in}\ triplet_{n-1}\ \texttt{cross}\ var_n\ \texttt{in}\ triplet_n\ := \ expr_1;\ \texttt{else}\ := \ expr_2], \end{array}$$

где $var_1, \ldots, var_n$ являются переменными, $triplet_1, \ldots, triplet_n$ являются триплетами, $expr_1$ является замещающим выражением, которое может зависеть от этих переменных, в то время как выражение по умолчанию $expr_2$ не зависит от них.

Такое выражение создает n-мерный массив с измерениями, соответствующими $hight_1, \ldots, hight_n$. Если индекс элемента массива принадлежит декартовому произведению триплетов, то значением элемента задается выражением $expr_1$. Иначе, его значение задается выражением $expr_2$.

Выражение замещения элементов массива имеет следующий вид:

$$source_array[var_1 \texttt{ in } triplet_1 \texttt{ cross } var_2 \texttt{ in } triplet_2 \texttt{ cross } \ldots$$
$$var_{n-1} \texttt{ in } triplet_{n-1} \texttt{ cross } var_n \texttt{ in } triplet_n \ := \ expr]$$

где $source_array$ является именем исходного массива, $var_1, \ldots, var_n$ являются переменными, триплетами являются $triplet_1, \ldots,$ $triplet_n$, $expr$ является замещающим выражением (возможно зависящим от $v_{1..n}$). Выражение замещения элементов массива выполняет неявные итерации над диапазоном, получаемым в результате декартового произведения триплетов. Результатом этого выражения является новый массив, который совпадает с $source_array$ за исключением элементов с индексами из диапазона, чьи значения заменяются на значение замещающего выражения.

Обозначим как $array_rep_expr$ такое выражение замещения элементов массива с такими подвыражениями.

5.2. Язык Cloud-Sisal-kernel

Язык Cloud-Sisal-kernel был разработан как промежуточный язык верификации для дедуктивной верификации Cloud Sisal программ [130]. Этот язык включает базовые конструкции Cloud Sisal, такие как триплеты, циклические выражения, циклические выражения с условиями завершения и выражения замещения элементов массивов. Отметим, что данный язык не включает выражения конструирования элементов массивов. В языке Cloud-Sisal-kernel можно сконструировать массив, используя циклические выражения, управляемые диапазоном, и редукцию $array\ of$.

Для языка Cloud-Sisal-kernel была предложена аксиоматическая семантика. Отметим, что циклические выражения языка Cloud-Sisal-kernel являются финитными итерациями над списком кортежей, полученным декартовым произведением триплетов. Поэтому, к циклическим выражениям языка Cloud-Sisal-kernel был применен символический метод верификации финитных итераций.

Входной язык системы ACL2 [150] является аппликативным и строго функциональным диалектом языка Common Lisp. Так как Cloud Sisal также является функциональным языком, то для языка Cloud-Sisal-kernel предложена трансля-

ция на язык системы ACL2. Функция *sisal2acl2* реализует эту трансляцию [130]. Рассмотрим описание этой трансляции.

5.2.1. Трансляционная семантика языка Cloud-Sisal-kernel

Функция *sisal2acl2* конвертирует массивы языка Cloud-Sisal-kernel в списки ACL2.

Функция *sisal2acl2* транслирует триплеты Cloud-Sisal-kernel в применения функции *triplet*. Эта функция принимает в качестве аргументов нижнюю границу *low*, верхнюю границу *high* и шаг *step* и возвращает список значений арифметической прогрессии, заданной триплетом.

Функция *sisal2acl2* использует транслятор *range2acl2*. Он транслирует декартово произведение триплетов в применение функции *cartesian_product* в общем случае. Функция *cartesian_product* возвращает список кортежей. Отметим, что также используется функция *partition* для моделирования заголовка цикла в случае одиночного триплета. Функция *range2acl2* транслирует одиночный триплет в применение функции *partition*.

Тип структуры *environment_id* генерируется для моделирования контекста. Поля этой структуры соответствуют переменным контекста. Функция *create_environment_id* генерируется для упрощения создания объектов типа *environment_id*. Функции *context_variables*, *context2vector* и *context2string* позволяют получать информацию о переменных контекста от компилятора CPPS [105, 108, 168] и конвертировать список таких переменных в строковое представление.

Так как циклические выражения языка Cloud-Sisal-kernel являются финитными итерациями, то функция *sisal2acl2* транслирует такие выражения в рекурсивные функции, реализующие операции замены.

Для моделирования выражения замещения элементов массива генерируется функция *update_elements_id* (*id* является уникальным идентификатором). Как и исходное Cloud Sisal выражение, эта функция создает такой новый массив что элементы, индексы которых принадлежат диапазону, заменя-

ются на значение выражения, зависимого от индексов. Определение функции *update_elements_id* генерируется с помощью трансляции замещающего выражения на язык системы ACL2.

Для моделирования циклических выражений, управляемых диапазоном, генерируется функция *rep_id*, где *id* является уникальным идентификатором. Отметим, что в данном случае финитная итерация выполняется над списком кортежей из декартового произведения триплетов. В качестве функций *choo* и *rest* из определения символического метода верификации финитных итераций используются функции *car* и *cdr* соответственно. Определение функции *rep_id* генерируется с помощью трансляции тела цикла на язык системы ACL2.

Таким образом, для языка Cloud-Sisal-kernel задана трансляционная семантика. Опишем ее подробнее.

5.2.2. Трансляция редукций на язык системы ACL2

Рассмотрим функцию *reduct2acl2*, реализующую трансляцию редукций Cloud Sisal на язык системы ACL2. Данная функция принимает следующие аргументы: имя редукции, редуцируемое выражение на текущей итерации ($expr_1$) и значение цикла после предыдущей итерации ($expr_2$). Эта функция моделирует на языке системы ACL2 применение редукции к *expr_1* и $expr_2$.

Также при моделировании редукции используется вспомогательная функция *reduction_init*, которая принимает имя редукции и возвращает значение редукции по умолчанию. Эта функция используется при генерации определений функций *rep* вместе с функцией *reduct2acl2*.

Определим функции *reduct2acl2* и *reduction_init* для редукций вида:

- Если *reduction* — это `array of`, тогда

$$\begin{aligned} &reduct2acl2(\texttt{array of},\ expr_1,\ expr_2) &=& \ (cons\ expr_1\ expr_2) \\ &reduction_init(\texttt{array of}) &=& \ nil \end{aligned}$$

  Мы моделируем массивы Cloud Sisal с помощью списков языка системы ACL2. Поэтому, значение по умолчанию является пустой список и

изначальным значением $expr_2$ также является пустой список. Применение *cons* к редуцируемой последовательности добавляет новый элемент в голову списка.

- Если *reduction* — это `value of`, тогда

$$reduct2acl2(\texttt{value of},\ expr_1,\ expr_2) = expr_1$$

Мы моделируем такую редукцию, возвращая текущее редуцируемое выражение. Поэтому, в результате получается последнее значение редуцируемой последовательности.

Значение по умолчанию зависит от типа $expr_1$, чтобы функция *rep* всегда возвращала значения одного и того же типа.

- Если *reduction* — это `sum of` *expr*, тогда

$$\begin{array}{lcl} reduct2acl2(\texttt{sum of},\ expr_1,\ expr_2) & = & (+\ expr_1\ expr_2) \\ reduction_init(\texttt{sum of}) & = & 0 \end{array}$$

Значением по умолчанию является 0, поэтому 0 является также начальным значением $expr_2$.

- Если *reduction* — это `product of` *expr*, тогда

$$\begin{array}{lcl} reduct2acl2(\texttt{productof},\ expr_1,\ expr_2) & = & (*\ expr_1\ expr_2) \\ reduction_init(\texttt{product of}) & = & 1 \end{array}$$

Значением по умолчанию является 1, поэтому 1 является также начальным значением $expr_2$.

- Если *reduction* — это `greatest of` *expr*, тогда

$$reduct2acl2(\texttt{greatest of},\ expr_1,\ expr_2) = (max\ expr_1\ expr_2)$$

Значение по умолчанию этой редукции зависит от наименьшего целого числа, представимого в разрядной сетке. Компилятор системы CPPS определяет такое значение.

- Если *reduction* — это `least of` *expr*, тогда

$$reduct2acl2(\texttt{least of},\ expr_1,\ expr_2) = (min\ expr_1\ expr_2)$$

  Значение по умолчанию этой редукции зависит от наибольшего целого числа, представимого в разрядной сетке. Компилятор системы CPPS определяет такое значение.

5.2.3. Трансляция базовых выражений Cloud Sisal на язык системы ACL2

Рассмотрим определение рекурсивной функции $sisal2acl2$ для базовых выражений языка Cloud Sisal.

Во-первых, рассмотрим трансляцию литералов и переменных.

Если $base_expr$ — это либо $false$ или $true$, тогда

$$sisal2acl2(base_expr) = boolean$$

где $boolean$ — это nil или t соответственно.

Если $base_expr$ — это десятичное целое или переменная, тогда

$$sisal2acl2(base_expr) = base_expr$$

Во-вторых, рассмотрим трансляцию условных выражений.

Если $if_expr \equiv$ **if** $cond$ **then** p_branch **else** n_branch **end if**, тогда

$$sisal2acl2(if_expr) = (if\ sisal2acl2(cond)\ sisal2acl2(p_branch)\ sisal2acl2(n_branch))$$

В-третьих, рассмотрим трансляцию бинарных операций.

Если $binary_expr \equiv arg1\ f\ arg2$ и $f \in \{+, -, *, /, <, >, <=, >=, =, \&, |, \hat{}\}$, тогда

$$sisal2acl2(binary_expr) = (f'\ sisal_to_acl2(arg1)\ sisal_to_acl2(arg2))$$

где f' — это строка из множества
$\{+, -, *, floor, <, >, <=, >=, equal, and, or, xor\}$ в зависимости от исходного f.

В-четвертых, рассмотрим трансляцию операций доступа к значению элемента массива.

Если $index_expr \equiv a[index_1,\ index_2, \ldots,\ index_n]$, где a — это n-мерный массив, $index_1, \ldots index_n$ являются Cloud Sisal выражениями, тогда

$$
\begin{array}{l}
sisal2acl2(index_expr)\ = \\
(nth\ sisal2acl2(index_n)\ (nth\ sisal2acl2(index_{n-1}) \\
\quad \ldots \\
\qquad (nth\ sisal2acl2(index_2)\ (nth\ sisal2acl2(index_1)\ a)\ \ldots\ ))
\end{array}
$$

В-пятых, рассмотрим трансляцию триплетов.

Если $triplet_expr$ — это триплет вида $[low\ ..\ hight\ ..\ step]$, тогда

$$sisal2acl2(triplet_expr) = (triplet\ sisal2acl2(low)\ sisal2acl2(hight)\ sisal2acl2(step))$$

В-шестых, рассмотрим трансляцию диапазонов.

Если $range_expr = triplet_1\ \texttt{cross}\ triplet_2\ \texttt{cross}\ \ldots\ \texttt{cross}\ triplet_n$, тогда определим трансляцию по индукции.

Если $n = 1$, тогда

$$range2acl2(range_expr)\ =\ (partition\ sisal2acl2(triplet_1))$$

Если $n > 1$, тогда

$$
\begin{array}{l}
range2acl2(range_expr)\ = \\
(cartesian_product\ sisal2acl2(triplet_1) \\
\qquad \ldots \\
\qquad\quad (cartesian_product\ sisal2acl2(triplet_{n-1})\ sisal2acl2(triplet_n))\ \ldots\ ))
\end{array}
$$

5.2.4. Алгоритм генерации структуры, хранящей переменные контекста

Пусть *context* обозначает множество имен переменных контекста. Алгоритм принимает его в качестве параметра и выполняет два действия. Во-пер-

вых, он создает тип структуры $environment_id$, чьи поля соответствуют переменным контекста. Во-вторых, он задает функцию $create_environment_id$, которая создает объект типа $environment_id$ с соответствующими значениями полей.

Сгенерированный код $create_environment_id$ выглядит следующим образом:

$$
\begin{array}{l}
(defun\ create_environment_id \\
\quad (\quad context2vector(context)_1 \\
\qquad \ldots \\
\qquad context2vector(context)_{context_length(context2vector(context))}) \\
(make\text{-}environment_id \\
\quad :context2vector(context)_1 \\
\qquad context2vector(context)_1 \\
\quad \ldots \\
\quad :context2vector(context)_{context_length(context2vector(context))} \\
\qquad context2vector(context)_{context_length(context2vector(context))}))
\end{array}
$$

Определением функции является применение макроса $make\text{-}environment_id$. Переменные контекста соответствуют полям возвращаемой структуры.

5.2.5. Алгоритм генерации функции *rep*

Данный алгоритм зависит от следующих параметров: 1) $var_1, \ldots, var_n$ — это переменные диапазона; 2) $expr$ — это выражение, которое может зависеть от данных переменных; 3) $reduction$ — это вид применяемой редукции. Зададим множество $context$ как

$$context = context_variables(expr) \setminus \{var_1, \ldots, var_n\}$$

В результате вычитания множеств получается множество переменных контекста без переменных диапазона.

Определение функции rep_id генерируется в следующем виде

$(defun\ rep_id\ (range_tuples\ environment)$

$\quad (b*\ ((when\ (endp\ range_tuples))\ reduction_init(reduction))$

$\qquad (tuple\ (car\ range_tuples))$

$\qquad (var_1\ (car\ tuple))$

$\qquad \ldots$

$\qquad (var_n\ (car\ (cdr\ (cdr\ \ldots\ (cdr\ tuple)\ \ldots\ ))))$

$\qquad (context2vector(context)_1$

$\qquad\quad environment.context2vector(context)_1)$

$\qquad \ldots$

$\qquad (context2vector(context)_{context_length(context2vector(context))}$

$\qquad environment.context2vector(context)_{context_length(context2vector(context))}))$

$\quad\ reduct2acl2(reduction,$

$\qquad sisal2acl2(expr),$

$\qquad (rep_id\ (cdr\ range_tuples)))))$

Связывание переменных $var_1, \ldots,\ var_n$ в блоке $b*$ позволяет использовать результат трансляции выражения $expr$, которое может зависеть от этих переменных. Для этого блок $b*$ связывает переменные контекста с полями объекта, содержащего контекст. Такой объект, называемый $environment$, создается функцией $create_environment_id$, которая применяется к переменным контекста.

Если список кортежей значений переменных диапазона $range_tuples$ является пустым, то функция rep_id возвращает значение редукции по умолчанию, полученное применением $reduction_init$ к имени редукции. Иначе, функция rep_id выполняет итерации над списком кортежей значений переменных диапазона, и для каждого кортежа она возвращает применение функции, моделирующей редукцию, к $expr$ и к рекурсивному вызову, моделирующему предыдущую итерацию. Имя функции, моделирующей редукцию, получается путем применения $reduct2acl2$ к имени редукции.

5.2.6. Алгоритм генерации функции *update_elements_id*

Параметры этого алгоритма совпадают с параметрами алгоритма из раздела 5.2.5 за исключением редукции. Множество *context* также задано по аналогии.

Определение функции *update_elements_id* генерируется в следующем виде

$(defun\ update_elements_id\ (indices_tuples\ environment\ source_array)$

$\quad (b*\ ((when\ (endp\ indices_tuples))\ source_array)$

$\qquad\quad (indices\ (car\ indices_tuples))$

$\qquad\quad (var_1\ (car\ indices))$

$\qquad\quad \ldots$

$\qquad\quad (var_n\ (car\ (cdr\ (cdr\ \ldots\ (cdr\ indices)\ \ldots\ ))))$

$\qquad (context2vector(context)_1$

$\qquad\quad environment.context2vector(context)_1)$

$\qquad \ldots$

$\qquad (context2vector(context)_{context_length(context2vector(context))}$

$\qquad environment.context2vector(context)_{context_length(context2vector(context))}))$

$\quad (update\text{-}nth\ var_1$

$\quad \ldots$

$\quad (update\text{-}nth\ var_n\ sisal2acl2(expr)$

$\qquad (nth\ var_{n-1}$

$\qquad \ldots$

$\qquad (nth\ var_1$

$\qquad (update_elements_id\ (cdr\ indices_tuples)\ source_array)))\ \ldots))))$

Определение функции *update_elements_id* также схоже с определением функции *rep_id* за исключением результирующего выражения в блоке $b*$ с применениями выражений *update-nth* вместо редуцирующей функции.

5.2.7. Трансляция циклических выражений Cloud-Sisal-kernel на язык системы ACL2

Функция *sisal2acl2* транслирует циклические выражения, управляемые диапазоном, в применение функции *rep_id*:

$$sisal2acl2(loop_e) \;=\; (rep_id$$
$$(reverse\ range2acl2(triplet_1\ cross\ \ldots\ triplet_{n-1}\ cross\ triplet_n))$$
$$(create_environment_id\ context2string\ context2vector($$
$$context_variables(expr) \setminus \{var_1,\ var_2,\ \ldots\ var_{n-1},\ var_n\})))),$$

где *loop_e* определено в разделе 5.1.1. Применение функции *reverse* гарантирует корректный порядок применения редукции. Операция разности множеств удаляет из контекста переменные диапазона. Строковые представления переменных становятся аргументами функции *create_environment_id*.

Функция *sisal2acl2* транслирует выражения замещения элементов массива в применение рекурсивных функций *update_elements_id*:

$$sisal2acl2(array_rep_expr) =$$
$$(update_elements_id$$
$$(reverse\ range2acl2(triplet_1\ cross\ \ldots\ cross\ \ldots\ triplet_{n-1}\ cross\ triplet_n))$$
$$(create_environment_id$$
$$context2string($$
$$context2vector($$
$$context_variables(expr) \setminus \{var_1,\ var_2,\ \ldots\ var_{n-1},\ var_n\})))$$
$$source_array),$$

где *array_rep_expr* определено в разделе 5.1.1. Отметим, что применение функции *update_elements_id* похоже на применение функции *rep_id* в случае циклических выражений.

5.2.8. Аксиоматическая семантика языка Cloud-Sisal-kernel

Для задания аксиоматической семантики языка Cloud-Sisal-kernel используется метод слабейшего предусловия [130]. Данный метод, как и символиче-

ский метод верификации финитных итераций, основан на замене переменных их значениями, вычисленными символически. Для задания аксиоматической семантики язык спецификаций был расширен термом $result$. Данный терм можно использовать в постусловии Sisal-выражения для обозначения значения этого выражения. Пусть $R(y \leftarrow expr)$ обозначает одновременную замену в R всех свободных вхождений переменной y на на $expr$. Если $expr$ является выражением языка Sisal, тогда $wp(expr, Q) = Q(result \leftarrow sisal2acl2(expr))$

Так как слабейшее предусловие Sisal-выражений основано на применении транслятора $sisal2acl2$, то аксиоматическая семантика языка Cloud-Sisal-kernel основана на трансляционной семантике.

5.3. Модули системы CPPS для дедуктивной верификации Cloud Sisal программ

Для дедуктивной верификации Cloud Sisal программ в системе CPPS были реализованы два модуля: CSV1 (Cloud Sisal Verification 1) и CSV2 (Cloud Sisal Verification 2) [106]. Рассмотрим данные модули подробнее.

Рассмотрим подробнее модуль CSV1. Одним из целевых языков системы программирования для языка Cloud Sisal является язык C. Система CPPS включает кодогенератор из внутреннего представления Cloud Sisal программы на язык C [2]. Данный транслятор генерирует код на подмножестве языка C, укладывающемся в язык C-light. Поэтому, было принято решение использовать систему C-lightVer для верификации C-представления Cloud Sisal программ [4]. Т.е. C-light является промежуточным языком верификации в данном подходе [106]. Отметим, что циклические выражения языка Cloud Sisal транслируются в циклы for, соответствующие финитным итерациям. Напомним, что исходные циклические выражения Cloud Sisal программы являются финитными итерациями. Такой результат трансляции финитных итераций позволяет применить методы комплексного подхода, реализованные в системе C-lightVer [125, 126]. Итого, модулем CSV1 является система C-lightVer.

Рассмотрим подробнее модуль CSV2. Отметим недостатки использования

модуля CSV1. Во-первых, использование промежуточного языка верификации может приводить к генерации более сложных УК. Во-вторых, модуль CSV1 зависит от продолжения разработки и поддержки транслятора на язык C. В-третьих, в случае наличия ошибки пользователю приходится анализировать УК не исходной Cloud Sisal программы, а УК промежуточной C-light программы, которая может значительно отличаться от исходной. Для решения этих проблем был разработан язык Cloud-Sisal-kernel [4]. Это представительное подмножество языка Cloud Sisal, для которого была разработана аксиоматическая семантика [130]. Cloud-Sisal-kernel программа, являющаяся одним из выходных данных системы CPPS, поступает на вход модулю CSV2. Модуль CSV2 является классической системой дедуктивной верификации, основанной на аксиоматической семантике Cloud-Sisal-kernel программ. Так как циклические выражения языка Cloud-Sisal-kernel соответствуют финитным итерациям, то модуль CSV2 позволяет проводить дедуктивную верификацию без использования инвариантов циклов.

Схема модулей системы CPPS для дедуктивной верификации Cloud Sisal программ изображена на рис. 5.1.

Рассмотрим следующие модули:

- Модуль CSV1 принимает на вход аннотированную C-light программы, которая является промежуточным представлением входной Cloud Sisal программы. Данный модуль производит дедуктивную верификацию данной программы, используя методы комплексного подхода.

- Модуль CSV2 состоит из следующих модулей:

  1. Модуль генерации УК. Модуль генерации УК порождает формулы, из истинности которых следует корректность программы. Если программа содержит циклические выражения, то данный модуль генерирует операции замены для них.

  2. Модуль управления доказательством УК применяет стратегии доказательства УК и генерирует леммы, соответствующие данным стра-

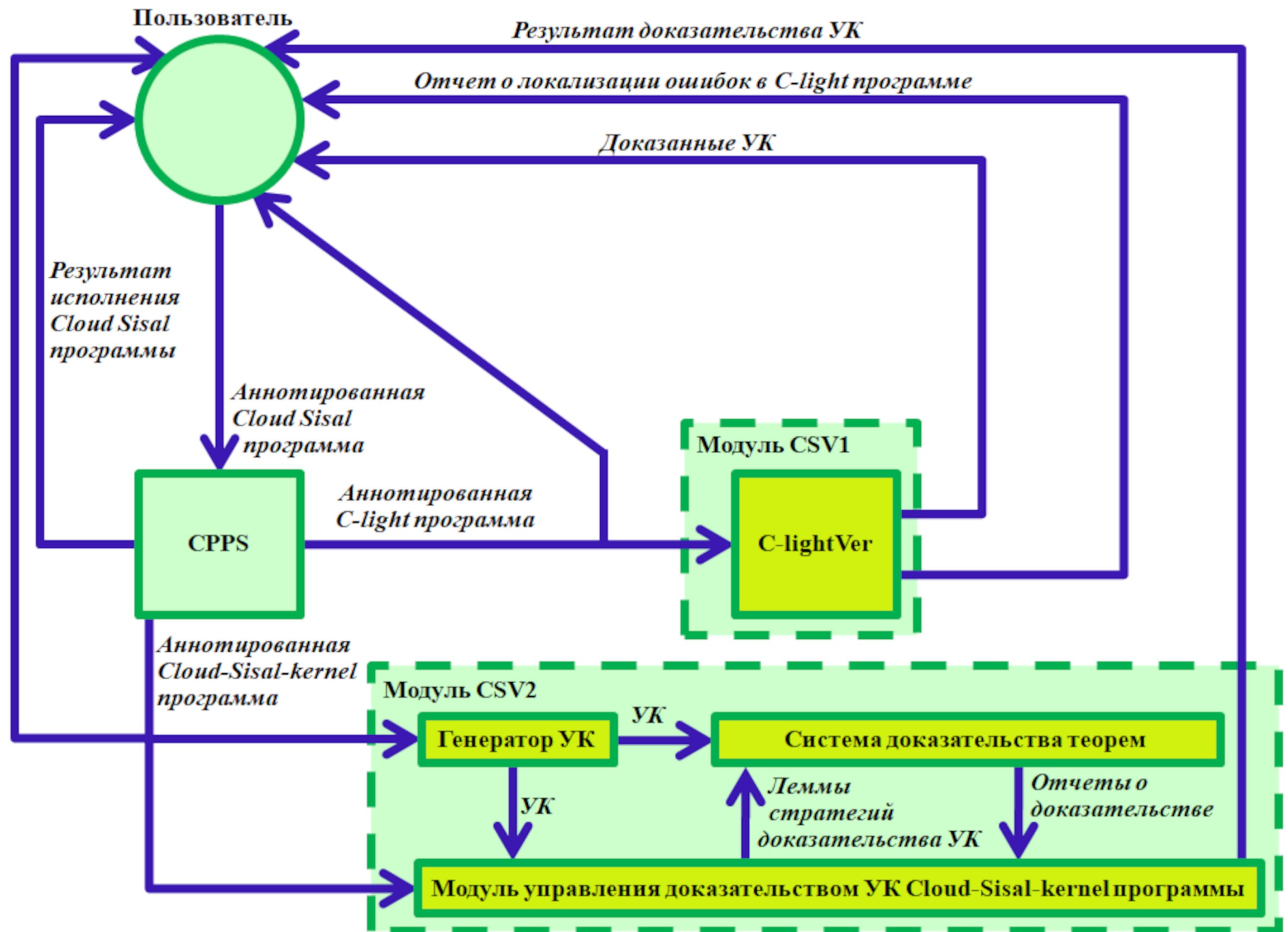

Рис. 5.1. Модули системы CPPS для дедуктивной верификации Cloud Sisal программ

тегиям. Леммы передаются на вход модулю доказательства теорем.

3. Модуль доказательства теорем проверяет истинность УК и лемм. В качестве данного модуля мы используем систему ACL2 [150].

Рассмотрим этапы дедуктивной верификации Cloud Sisal программ с помощью модуля CSV1:

1. Аннотированная Cloud Sisal программа поступает на вход системе CPPS.

2. Системе CPPS транслирует внутреннее представление аннотированной Cloud Sisal программы в аннотированную C-light программу.

3. Полученная аннотированная C-light программа передается пользователю и поступает на вход системы C-lightVer.

4. Система C-lightVer проводит автоматизированную дедуктивную верификацию полученной аннотированной C-light программы, используя методы

комплексного подхода.

5. Пользователю передаются результаты исполнения системы C-lightVer в виде доказанных УК. Если все УК доказаны, то дедуктивная верификация завершается.

6. Пользователю передаются результаты исполнения системы C-lightVer в виде отчета о локализации ошибок в C-light программе в случае недоказанных УК. Пользователь может анализировать аннотированную C-light программу и отчет, чтобы проверить наличие ошибок в исходной Cloud Sisal программе.

Рассмотрим этапы дедуктивной верификации Cloud Sisal программ с помощью модуля CSV2:

1. Аннотированная Cloud Sisal программа поступает на вход системе CPPS.

2. Система CPPS возвращает аннотированную Cloud-Sisal-kernel программу.

3. Полученная аннотированная программа на языке Cloud-Sisal-kernel передается пользователю и поступает на вход генератора УК и модуля управления доказательством УК.

4. Генерируются УК полученной Cloud-Sisal-kernel программы.

5. Система доказательства теорем проверяет истинность УК. Доказанные УК передаются пользователю. Если все УК доказаны, то дедуктивная верификация завершается.

6. Модуль управления доказательством последовательно применяет все подходящие стратегии для доказательства УК. Их применение состоит в генерации и доказательстве лемм. Доказанные леммы добавляются в теорию предметной области, которая используется в ACL2. Если применение стратегий привело к доказательству всех УК, то дедуктивная верификация завершается.

7. Пользователю передаются недоказанные УК. Пользователь может анализировать аннотированную Cloud-Sisal-kernel программу, используя недоказанные УК, чтобы проверить наличие ошибок в исходной Cloud Sisal программе.

Реализация модулей CSV1 и CSV2 позволяет не только избежать задания инвариантов циклических выражений, но и автоматизировать доказательство УК с операциями замены.

5.4. Расширение языка C конструкциями языка Sisal, позволяющее применять в системе CPPS методы комплексного подхода системы C-lightVer

В рамках проекта РНФ № 18-11-00118 «Облачные методы и средства конструирования эффективных и надежных параллельных программ на основе функциональных спецификаций и семантических преобразований» важной задачей являлось автоматическое распараллеливание такого надмножества C, которое позволяет реализовать автоматическую верификацию. Предложенным решением является такое надмножество C-kernel, как C-Sisal-kernel [15]. Это расширение C-kernel циклами языка Cloud-Sisal-kernel. Реализация неявного параллельного исполнения программ на языке C-Sisal-kernel основано на опыте [105, 108, 168] распараллеливания программ на языке Cloud Sisal. Отметим, что это задача другой группы исследователей [105, 108, 168, 176] из Института систем информатики СО РАН. Эта группа разрабатывает CPPS. Автоматическое распараллеливание C-Sisal-kernel программ является частью проекта CPPS.

Задачей в рамках проекта C-lightVer являлось расширение аксиоматической семантики языка C-kernel для реализации дедуктивной верификации C-программ с циклами языка Sisal. Для решения этой задачи применялись методы комплексного подхода. Дедуктивная верификация C-Sisal-kernel программ является частью системы C-lightVer.

5.4.1. Синтаксис триплетов, циклических выражений, циклических выражений с условиями завершения и выражений замещения элементов массивов в языках C-light и C-kernel

Язык C-light был расширен таким новым видом выражений, как триплет. Триплет имеет следующий синтаксис: lower boundary .. upper boundary .. step, где *lower boundary*, *upper boundary* и *step* – целочисленные выражения. Рассмотрим разницу между операционной семантикой этих выражений в языке C-light и семантикой триплетов Cloud-Sisal-kernel.

Триплеты, добавленные в язык C-light, возвращают структуру типа *tripl*:

typedef struct tripl{**int scalar**; **int** ∗ **vector**; }**tripl**;

Так как триплет моделирует последовательность чисел, то поле *vector* этой структуры является указателем на динамический массив, заполненный этой последовательностью (выделенная память должна быть освобождена программистом после использования), поле *scalar* хранит длину полученного массива. Пустой триплет соответствует структуре, где поле *scalar* равно 0 и поле *vector* равно $NULL$.

Также язык C-light был расширен циклическими выражениями с синтаксисом, сходным с синтаксисом *loop_e*, где *loop_e* определено в разделе 5.1.1. Опишем разницу между семантикой этих выражений в языке C-light и семантикой циклов Cloud-Sisal-kernel. Циклические выражения, добавленные в язык C-light, возвращают структуру типа *loop_expr*:

typedef struct loop_expr{**int scalar**; **int** ∗ **vector**;}**loop_expr**;

Если циклическое выражение возвращает целочисленное значение, тогда результат сохраняется в поле *scalar* соответствующей структуры и поле *vector* хранит $NULL$. Если циклическое выражение возвращает целочисленный массив, то поле *vector* рассматриваемой структуры является указателем на динамический массив, соответствующий результату (выделенная память должна быть освобождена программистом после использования), длина полученного массива хранится в поле *scalar*.

Отметим, что определение структуры *loop_expr* совпадает с определением структуры *tripl*. Поэтому, возможно использовать единую структуру для хранения триплетов и результатов циклических выражений. Но было принято решение использовать для хранения триплетов и результатов циклических выражений структуры с разными наименованиями. Это решение является шагом на пути к улучшению типизации в расширении языка C-light и в расширении языка C-kernel. Также это решение может упростить программирование на данных языках.

Также язык C-light был расширен циклическими выражениями с условиями завершения. В качестве условий завершения используются выражения языка C-light.

Также язык C-light был расширен выражениями замещения элементов массивов с синтаксисом, сходным с *array_rep_expr*, где *array_rep_expr* определено в разделе 5.1.1. Но эти выражения замещения элементов массивов были реализованы как инструкции языка C-light. Поэтому, будем называть их инструкциями замещения элементов массивов.

Язык, полученный расширением языка C-light конструкциями языка Cloud-Sisal-kernel, называется C-Sisal-light [15]. Язык C-kernel также был расширен конструкциями языка Cloud-Sisal-kernel, в которые и транслируются соответствующие конструкции C-Sisal-light. Но подвыражения этих конструкций, например, редуцируемые выражения циклов, могут измениться в результате трансляции.

Так как триплеты, циклические выражения и выражения замещения элементов массивов могут содержать побочные эффекты (например, выделение динамической памяти), то правила трансляции, которые локализуют побочные эффекты (например, правило *Ops*), были модифицированы. Триплеты, циклические выражения и выражения замещения элементов массивов были добавлены в область действия этих правил. В результате редуцируемые подвыражения циклов транслируются из C-Sisal-light в C-kernel. Также транслятор *c2acl2* из C-kernel в ACL был модифицирован для трансляции триплетов и циклических выражений.

Язык, полученный расширением языка C-kernel конструкциями языка Cloud-Sisal-kernel, называется C-Sisal-kernel.

5.4.2. Семантика триплетов, циклических выражений, циклических выражений с условиями завершения и выражений замещения элементов массивов в языке C-Sisal-kernel

Аксиоматическая семантика языка C-kernel была расширена правилами вывода для триплетов, циклических выражений и инструкций замещения элементов массивов [15]. Все инструкции, содержащие триплеты, имеют вид следующего присваивания из-за применения правил трансляции, локализующих побочные эффекты: $var\ =\ low\ ..high\ ..step;$, где var является переменной типа $tripl$, low, $high$ и $step$ являются целочисленными переменными или константами. Эта инструкция является присваиванием триплета переменной. Рассмотрим правило вывода для этой инструкции:

$$\frac{\{P\}\ \mathbf{A};\ \{Q(var\ \leftarrow c2acl2(low\ ..hight\ ..step),\ MD \leftarrow upd(MD, var.vector, c2acl2(low\ ..high\ ..step).vector))\}}{\{P\}\ \mathbf{A}; \mathbf{var}\ =\ \mathbf{low}\ ..\mathbf{high}\ ..\mathbf{step};\ \{Q\}}$$

где A являются инструкциями программы до рассматриваемого присваивания. Мы используем обратное прослеживание (метод слабейшего предусловия [38]): мы движемся от конца программы к ее началу и элиминируем самый правый оператор (на верхнем уровне), применяя соответствующее правило вывода смешанной аксиоматической семантики [147] языка C-kernel. Отметим, что новое постусловие получается из исходного Q одновременной заменой var на значение триплета и заменой MD на новое состояние памяти. Структура $tripl$ была задана в ACL2, используя конструкции библиотеки fty системы ACL2. Транслятор $c2acl2$ генерирует применение конструктора структуры $tripl$, который устанавливает в качестве значения поля $vector$ значение функции $triplet$.

Все инструкции, содержащие циклические выражения, имеют вид следующего присваивания из-за применения правил трансляции, которые локализуют побочные эффекты: $var\ =\ loop_e;$. Эта инструкция является присваиванием

циклического выражения переменной. Рассмотрим следующее правило вывода для этой инструкции:

$$\frac{\{P\}\ \mathbf{A};\ \{Q(var\ \leftarrow c2acl2(loop_e),\ MD \leftarrow upd(MD, var.vector, c2acl2(loop_e).vector))\}}{\{P\}\ \mathbf{A};\ \mathbf{var}\ =\ \mathbf{loop_e}\ \{Q\}}$$

где A определяется по аналогии с правилом для триплетов. Отметим, что структура $loop_expr$ была задана на языке ACL, используя конструкции библиотеки fty системы ACL2. Эта структура аналогична структуре $loop_expr$, заданной на языке C-light. Определение транслятора $c2acl2$ для этого выражения основано на модификации транслятора $sisal2acl2$ для циклических выражений:

$c2acl2(loop_e)\ =\ (make\text{-}loop_expr$
$:scalar\ (rep_id$
$(reverse\ range2acl2(triplet_1\ cross\ \ldots\ cross\ triplet_n))$
$(create_environment_id\ context2string(context2vector($
$context_variables(expr) \setminus \{var_1,\ var_2,\ \ldots\ var_{n-1},\ var_n\}))))$
$:vector\ nil)$

где id является уникальным идентификатором. Это результат трансляции в случае целочисленного значения редукции. Если редукция возвращает целочисленный массив, то результат редукции связывается с полем $vector$ вместо поля $scalar$, поле $scalar$ связывается с длиной результата редукции. Результатом трансляции является использование конструктора структуры $loop_expr$, который задает значения полей, используя значение функции rep. Рассмотрим алгоритм генерации определения функции rep_id:

$(defun\ rep_id\ (range_tuples\ environment)$
$(b*\ ((when\ (endp\ range_tuples))\ reduction_init(reduction))$
$(tuple\ (car\ range_tuples))$
$(var_1\ (car\ tuple))$
$\ldots$

$(var_n\ (car\ (cdr\ (cdr\ \ldots\ (cdr\ tuple)\ \ldots\ ))))$

$(previous_iter_result\ (rep_id\ (cdr\ range_tuples)\ environment)))$

$reduct2acl2(reduction,\ c2acl2(expr),\ previous_iter_result)))$

где $range_tuples$ является списком, соответствующим результату декартового произведения триплетов. Каждый элемент этого списка является кортежом значений переменных заголовка цикла. Редукция реализована, используя результат транслятора $reduct2acl2$. Применения car и cdr к $range_tuples$ соответствуют применениям $choo$ и $rest$ к S в определении символического метода верификации финитных итераций. Это демонстрирует, что циклические выражения являются финитными итерациями над декартовым произведением триплетов. Главным изменением этого алгоритма относительно генерации rep_id для циклов Cloud-Sisal-kernel [130] является применение транслятора $c2acl2$ вместо функции $sisal2acl2$.

Метод смешанной аксиоматической семантики позволяет задать версию правила вывода для циклических выражений в случае скалярного результата и отсутствия выделения динамической памяти. Запишем эту версию правила вывода на языке шаблонов [124] для использования этого правила в качестве входа метагенератора УК:

```
{P} A {substitution(Q, var, c2acl2(
   for int_var(var_element_1) in triplet_val(triplet_element_1)
   repeat(cross int_var(var_element_2) in triplet_val(triplet_element_2))
   returns reduction int_val(expr) end for)}
|- {any_predicate(P)} any_code(A)
var = for int_var(var_element_1) in triplet_val(triplet_element_1)
   repeat(cross int_var(var_element_2) in triplet_val(triplet_element_2))
   returns reduction int_val(expr) end for {any_predicate(Q)}
```

где `repeat` является конструкцией, которая позволяет задавать шаблоны, основанные на повторах, `elements` с индексами [126] является конструкцией, которая позволяет задавать векторы переменных и выражений одинакового типа. Конструкция `int_var` сопоставляется с целочисленной переменной, конструкция `int_val` сопоставляется с целочисленным значением и конструкция

`triplet_val` сопоставляется с триплетом. Правило вывода для присваивания триплету в случае пустого триплета и отсутствия выделения динамической памяти является схожим с рассмотренным правилом.

Правило вывода для циклического выражения с условием завершения является таким же, как правило вывода для циклического выражения. Главным отличием является алгоритм генерации определения функции *rep_id*, расширенный генерацией проверки условия завершения и генерацией проверки результата предыдущей итерации. Рассмотрим генерацию определения функции *rep_id*:

$$
\begin{array}{l}
(defun\ rep_id\ (range_tuples\ environment) \\
\ \ (b* \\
\ \ \ \ (when\ (endp\ range_tuples)) \\
\ \ \ \ (update\ \ :loop\text{-}break\ nil\ reduction_init(reduction))) \\
\ \ \ \ (tuple\ (car\ range_tuples)) \\
\ \ \ \ (var_1\ (car\ tuple)) \\
\ \ \ \ \ldots \\
\ \ \ \ (var_n\ (car\ (cdr\ (cdr\ \ldots\ (cdr\ tuple)\ \ldots\ )))) \\
\ \ \ \ (previous_iter_result\ (rep_id\ (cdr\ range_tuples)\ environment)) \\
\ \ \ \ ((when\ (not\ previous_iter_result.loop\text{-}break))\ previous_iter_result) \\
\ \ \ \ ((when\ (not\ c2acl2(condition))) \\
\ \ \ \ (update\ \ :loop\text{-}break\ t\ previous_iter_result))) \\
\ \ (update\ \ :loop\text{-}break\ nil \\
\ \ \ \ reduct2acl2(reduction,\ c2acl2(expr),\ previous_iter_result))))
\end{array}
$$

где мы связываем *previous_iter_result* с результатом предыдущей итерации. Если список кортежей диапазона равен *nil*, тогда значением функции является структура со значением редукции по умолчанию и со значением поля *loop-break*, которое равно $false$. Если значение поля *loop-break* из результата предыдущей итерации равно *true*, тогда значением этой функции является *previous_iter_result*. Если условие завершения выполнено, то значением функции является результат предыдущей итерации за исключением значения поля

loop-break, которое равно *true*. Эти проверки реализованы в первой, во второй и в третьей конструкциях *when* соответственно. Структура с результатом применения редукции и со значением поля *loop-break*, равным $false$, является результатом этой функции в других случаях.

Рассмотрим правило вывода для инструкции замещения элементов массива:

$$\frac{\{P\}\ \mathbf{A};\ \{Q(source_array\ \leftarrow c2acl2(array_rep_expr),\quad MD \leftarrow upd(MD, source_array, c2acl2(array_rep_expr)))\}}{\{P\}\ \mathbf{A};\ \mathbf{array_rep_expr};\ \{Q\}}$$

где A определено по аналогии с правилом для триплетов. Определение транслятора $c2acl2$ для этой инструкции схоже с определением транслятора $sisal2acl2$ для выражений замещения элементов массивов. Рассмотрим алгоритм генерации определения функции $update_elements_id$ в случае языка C-Sisal-kernel:

$(defun\ update_elements_id\ (range_tuples\ environment\ source_array)$
$\quad (b*\ ((when\ (endp\ range_tuples))\ source_array)$
$\qquad\quad (tuple\ (car\ range_tuples))$
$\qquad\quad (var_1\ (car\ tuple))$
$\qquad\quad \ldots$
$\qquad\quad (var_n\ (car\ (cdr\ (cdr\ \ldots\ (cdr\ tuple)\ \ldots\ )))))$
$\qquad (update\text{-}nth\ var_1$
$\qquad \ldots$
$\qquad (update\text{-}nth\ var_{n-1}$
$\qquad (update\text{-}nth\ var_n\ c2acl2(expr)$
$\qquad\quad (nth\ var_{n-1}$
$\qquad\quad \ldots$
$\qquad\quad (nth\ var_1$
$\qquad\qquad (update_elements_id(cdr\ range_tuples)\ environment$
$\qquad\qquad\quad source_array)))\ldots))))$

где $source_array$ является списком вложенных списков в случае многомерного массива. Мы используем вложенные применения функций *nth* и *update-nth* для

обновления элемента в многомерном массиве. Главным изменением этого алгоритма относительно генерации *update_elements_id* в случае Cloud-Sisal-kernel является применение транслятора *c2acl2* вместо функции *sisal2acl2*. Кроме того, мы задали это правило вывода на языке шаблонов [124] для использования этого правила в качестве входа метагенератора УК:

```
{P} A {simultaneous_substitution(Q, source_array, c2acl2(
   source_array[int_var(var_element_1) in triplet_val(triplet_element_1)
   repeat(cross int_var(var_element_2) in triplet_val(triplet_element_2))
   := int_val(expr)]),
   MD, (upd MD source_array c2acl2(
   source_array[int_var(var_element_1) in triplet_val(triplet_element_1)
   repeat(cross int_var(var_element_2) in triplet_val(triplet_element_2))
   := int_val(expr)]))))}
|- {any_predicate(P)} any_code(A)
source_array[int_var(var_element_1) in triplet_val(triplet_element_1)
   repeat(cross int_var(var_element_2) in triplet_val(triplet_element_1))
   := int_val(expr)] {any_predicate(Q)}
```

где `simultaneous_substitution` является конструкцией, которая позволяет выполнять одновременную замену. Наша реализация транслятора *c2acl2* и конструкции `triplet_val` в метагенераторе позволяет задавать новые правила вывода как входные данные системы C-lightVer. Данный подход упростил расширение системы верификации C-lightVer рассмотренными правилами вывода.

5.5. Эксперименты по верификации программ на языках Cloud Sisal, Cloud-Sisal-kernel и C-Sisal-kernel

Рассмотрим эксперименты по применению комплексного подхода к программам на языках Cloud Sisal, Cloud-Sisal-kernel и C-Sisal-kernel [4, 15, 123, 125, 130].

5.5.1. C-представление Sisal программы, проверяющей наличие в массиве не менее заданного количества вхождений ключа

Рассмотрим верификацию следующей программы на языке Cloud Sisal с помощью модуля CSV1 [125]:

```
function search_count (a: array of integer, n: integer, key: integer
                       returns integer)
    for cnt := 0, result := 0; i in 1..n
    while !(cnt = entr) do
        cnt := if a[i]=key then old cnt + 1;
        result := if cnt=entr then 1;
    returns value of result end for end function
```

Приведем предусловие данной программы (на языке системы ACL2):

```
(and (integer-listp a) (integerp n) (integerp key) (< 0 n) (<= n (length a)))
```

и постусловие:

```
(and (implies (=> entr (cnt 0 (- n 1) key a)) (equal result 1))
     (implies (< entr (cnt 0 (- n 1) key a)) (equal result 0)))
```

Функция `cnt` вычисляет количество элементов, равных ключу `key` в диапазоне от `i`-того до `j`-того элемента списка `a` (мы моделируем массивы с помощью списков). Приведем определение функции `cnt`:

```
(defun cnt(i j key a)
    (if (or (not (natp i)) (not (natp j))) 0
       (if (> i j) 0
            (if (= i j)
                (if (= key (nth j a)) 1 0)
                (if (= key (nth j a))
                    (+ 1 (cnt i (- j 1) key a))
                    (cnt i (- j 1) key a))))))
```

В результате трансляции данной программы была получена следующая C-light программа:

```
int search_count(int* arr, int length, int key, int entr) {
    auto int result = 0, cnt = 0, i;
    for (i = 0; i < length; i++) {
        if (key == arr[i]) cnt++;
        if (cnt == entr) {result = 1; break;}}
    return result;}
```

Верифицируем данную C-light программу для верификации исходной Sisal-программы, исходя из рассмотренного предусловия и постусловия.

Применим нашу систему для дедуктивной верификации такой программы. Условие корректности для нашей аннотированной программы было порождено автоматически системой C-lightVer. Рассмотрим данное УК:

```
(defthm my-theorem1
    (implies
        (and (integerp n) (integerp key) (integerp entr) (integer-listp a))
        (implies
            (and (< 0 n) (<= n (length a)) (< 0 entr))
            (and
                (implies
                    (<= entr (cnt 0 (- n 1) key a)) (= (rep2 n key entr a) 1))
                (implies
                    (> entr (cnt 0 (- n 1) key a)) (= (rep2 n key entr a) 0)))))
    :hints (("Goal" :induct (dec-induct n))))
```

Так как в программе содержится цикл, осуществляющий итерацию над массивом, то при генерации данного условия корректности был применен символический метод верификации финитных итераций. Поэтому, данное условие корректности основано на использовании порожденных автоматически функций `rep1` и `rep2`, представляющих собой операцию замены для изменяемых в теле цикла переменных.

Приведем определение функции `rep1`, выражающей в символической форме эффект от изменения в цикле переменной `cnt`:

```
(defun rep1(i key entr a)
    (if (zp i) 0
```

```
    (if (= entr (rep1 (- i 1) key entr a))
        (rep1 (- i 1) key entr a)
        (if (= key (nth (- i 1) a))
            (+ 1 (rep1 (- i 1) key entr a)) (rep1 (- i 1) key entr a)))))
```

и приведем определение функции `rep2`, выражающей в символической форме эффект от изменения в цикле переменной `result`:

```
(defun rep2(i key entr a)
    (if (zp i) 0
        (if (= entr (rep1 i key entr a)) 1 (rep2 (- i 1) key entr a))))
```

Операция замены всегда определяется с помощью рекурсии по первому аргументу, соответствующему номеру итерации. Поэтому, условия корректности, содержащие операцию замены, необходимо доказывать с помощью индукции по переменной длины структуры, над которой осуществляется итерация (`n` в рассматриваемом примере).

Отметим, что система ACL2 не справляется с доказательством этого примера, используя только индукцию по `n`. Такой ответ системы ACL2 был проанализирован в автоматическом режиме и в автоматическом режиме была запущена стратегия усиления УК, описанная в разделе 2.4.2. Отметим, что данная стратегия разбита на последовательность шагов (возможно с повторениями).

Обозначим A следующую формулу:

```
(and (integerp n) (integerp key) (integerp entr) (integer-listp a))
```

Эта формула является конъюнкцией утверждений о целочисленности переменных n, key и $entr$, а также утверждения о том, что a – список целочисленных элементов.

Обозначим B следующую формулу:

```
(and (< 0 n) (<= n (length a)) (< 0 entr))
```

Эта формула является конъюнкцией утверждений о том, что n больше, чем n, длина последовательности a больше или равна n, и $entr$ больше, чем 0.

Обозначим C следующую формулу: `(<= entr (cnt 0 (- n 1) key a))`. Эта формула является утверждением о том, что количество элементов в последовательности a от элемента с индексом 0 до элемента с индексом $n-1$ больше или равно $entr$.

Обозначим D следующую формулу: `(= (rep2 n key entr a) 1)`. Эта формула является утверждением о том, что значение переменной $result$ после исполнения цикла, которому соответствует вызов функции $rep2$ с аргументами n, key, $entr$ и a равно 1.

Обозначим E следующую формулу: `(> entr (cnt 0 (- n 1) key a))` (Эта формула является утверждением о том, что количество элементов в последовательности a от элемента с индексом 0 до элемента с индексом $n-1$ меньше, чем $entr$.)

Обозначим F следующую формулу: `(= (rep2 n key entr a) 0)`. Эта формула является утверждением о том, что значение переменной $result$ после исполнения цикла, которому соответствует вызов функции $rep2$ с аргументами n, key, $entr$ и a равно 0.

Обозначим J следующую формулу: `(= entr (rep1 n key entr a))`. Эта формула является утверждением о равенстве значения $entr$ и значения переменной cnt после исполнения цикла, которому соответствует вызов функции $rep1$ с аргументами n, key, $entr$ и a.

Обозначим K следующую формулу: `(zp n)`. Эта формула является утверждением о том, что n является целым числом, значение которого больше, чем 0.

Конъюнкцию $A \wedge B \wedge C$ обозначим как L, а конъюнкцию $A \wedge B \wedge E$ обозначим как M. Отметим, что данные формулы не отражают структуры данных, с которыми работает алгоритм, эти обозначения введены только для ясности изложения.

До применения шага 1 условие корректности соответствовало шаблону

$$A \Rightarrow (B \Rightarrow ((C \Rightarrow D) \wedge (E \Rightarrow F))).$$

Обозначим условие корректности как ϕ. Наша стратегия доказательства

из раздела 2.4.2 была применена для проверки ее истинности.

Шаги 1-4 являются применением шагов 1, 3, 4 и 1 соответственно. После применения шага 1 условие корректности было преобразовано в конъюнкцию клозов, соответствующую следующему шаблону $(L \Rightarrow D) \wedge (M \Rightarrow F)$. Обозначим полученную формулу формулой ϕ_1. Отметим, что на текущем этапе исполнения стратегии сохраняется эквивалентность между полученной формулой и условием корректности.

Применение шага 2 на текущем этапе исполнения стратегии не привело к преобразованию формулы ϕ_1. На шаге 3 новое, нерекурсивоное определение для функции `rep1` не было сгенерировано, при этом новое, нерекурсивное определение для функции `rep2` было сгенерировано автоматически. При этом новое определение эквивалентно исходному. Это стало следствием того, что автоматически системой ACL2 была доказана индукцией по переменной `n` следующая теорема: $(A \wedge B \wedge \neg J) \Rightarrow F$. Данная теорема является утверждением о том, что при невыполнении условия выхода из цикла J значение функции `rep2` совпадает с начальным значением переменной `result`, т.е. `0`. Таким образом, значение функции `rep2` при условии невыхода из цикла было переопределено как `0`. На стадии генерации операций замены было установлено, что функция `rep1`, вызываемая функцией `rep2`, не вызывает других функций, кроме самой себя. Поэтому, автоматически делается вывод о том, что функция `rep2` была переопределена нерекурсивным образом.

Рассмотрим следующее нерекурсивное определение этой функции `rep2`:

```
(defun rep2(i key entr a)
   (if (zp i) 0
      (if (= entr (rep1 i key entr a)) 1 0)))
```

В результате было порождено следующее дерево, представляющее нерекурсивное определение данной функции `rep2`.

Рисунок 5.2 является схемой дерева нерекурсивно определенной функции `rep2`. (Обозначим это дерево `rep2_tree`.) Пометка P ребра этого дерева обозначает формулу $(zp\ i)$, а пометка $\neg P$ обозначает, соответственно, отрицание

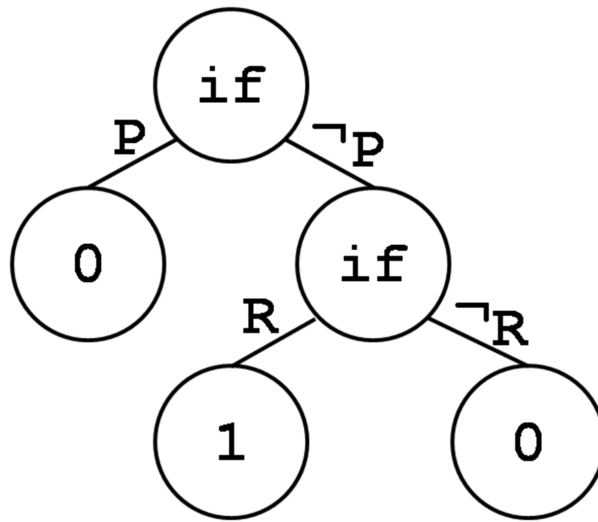

Рис. 5.2. Дерево, представляющее нерекурсивное определение функции `rep2`

этой формулы. Пометка R обозначает формулу $(=\ entr\ (rep1\ i\ key\ entr\ a))$, а пометка $\neg R$ обозначает, соответственно, отрицание этой формулы. Эта схема означает, что при выполнении условия P возвращаемым значением функции `rep2` является `0`. При невыполнении условия P и при выполнении условия R возвращаемым значением функции `rep2` является `1`. При невыполнении условия P и при невыполнении условия R возвращаемым значением функции `rep2` является `0`.

На следующем шаге 1 сначала была рассмотрена подформула D формулы ϕ_1, так как она содержит вызов нерекурсивной функции $rep2$. Дерево $rep2_tree_1$ получено из дерева $rep2_tree$ заменой всех переменных в пометках ребер на аргументы вызова $rep2$ в формуле D. Таким образом, новые пометки ребер P_1, $\neg P_1$, R_1 и $\neg R_1$ получены заменой в формулах P, $\neg P$, R и $\neg R$ вхождения i на n, key на key, $entr$ на $entr$, a на a. Таким образом, формула P_1 совпадает с K, формула $\neg P_1$ совпадает с $\neg K$, формула R_1 совпадает с J, формула R_1 совпадает с $\neg J_1$.

Формула ϕ_2 получена из ϕ_1 путем замены подформулы D на конъюнкцию специальных импликаций, построенных с помощью дерева $rep2_tree_1$. Рассмотрим формулу ϕ_2:

$$(L \Rightarrow$$
$$((K \Rightarrow (0 = 1)) \wedge ((\neg K \wedge J) \Rightarrow (1 = 1)) \wedge ((\neg K \wedge \neg J) \Rightarrow (0 = 1)))) \wedge (M \Rightarrow F)$$

Заключение каждой такой импликации получено путем замены в заключении D вызова функции замены $rep2$ на выражение, определяемое из посылки этой импликации. Первая такая импликация означает, что при выполнении условия K равенство 0 и 1 эквивалентно равенству вызова функции $rep2$ и 1. (Эта импликация соответствует пути от корня $rep2_tree_1$ по ребру P_1.) Вторая такая импликация означает, что при невыполнении условия K и при выполнении условия J равенство 1 и 1 эквивалентно равенству вызова функции $rep2$ и 1. (Эта импликация соответствует пути от корня $rep2_tree_1$ по ребру $\neg P_1$ и по ребру R_1.) Третья такая импликация означает, что при невыполнении условия K и при невыполнении условия J равенство 0 и 1 эквивалентно равенству вызова функции $rep2$ и 1. (Эта импликация соответствует пути от корня $rep2_tree_1$ по ребру $\neg P_1$ и по ребру $\neg R_1$.)

Далее была рассмотрена подформула D формулы ϕ_1. Дерево $rep2_tree_2$ получено из дерева $rep2_tree$ по аналогии с деревом $rep2_tree_1$. При этом, дерево $rep2_tree_2$ совпало с деревом $rep2_tree_1$.

Формула ϕ_3 получена из ϕ_2 путем замены подформулы F на конъюнкцию специальных импликаций, построенных с помощью дерева $rep2_tree_2$. Рассмотрим формулу ϕ_3:

$$\begin{gathered}(L \Rightarrow\\ ((K \Rightarrow (0=1)) \wedge ((\neg K \wedge J) \Rightarrow (1=1)) \wedge ((\neg K \wedge \neg J) \Rightarrow (0=1))))\wedge\\ (M \Rightarrow\\ ((K \Rightarrow (0=0)) \wedge ((\neg K \wedge J) \Rightarrow (1=0)) \wedge ((\neg K \wedge \neg J) \Rightarrow (0=0))))\end{gathered}$$

Формула ϕ_4 получена из формулы ϕ_3 путем замены импликаций на дизъюнкцию заключения и отрицание посылки. При этом, отрицания посылок были преобразованы по законам де Моргана:

$$\begin{gathered}(L \Rightarrow\\ ((\neg K \vee (0=1)) \wedge ((K \vee \neg J) \vee (1=1)) \wedge ((K \vee J) \vee (0=1))))\wedge\\ (M \Rightarrow\\ ((\neg K \vee (0=0)) \wedge ((K \vee \neg J) \vee (1=0)) \wedge ((K \vee J) \vee (0=0))))\end{gathered}$$

Формула ϕ_5 получена из формулы ϕ_4 путем преобразования к форме конъюнкции дизъюнктов:

$$(L \Rightarrow (\neg K \vee (0 = 1))) \wedge (L \Rightarrow ((K \vee \neg J) \vee (1 = 1))) \wedge (L \Rightarrow ((K \vee J) \vee (0 = 1))) \wedge$$
$$(M \Rightarrow (\neg K \vee (0 = 0))) \wedge (M \Rightarrow ((K \vee \neg J) \vee (1 = 0))) \wedge (M \Rightarrow ((K \vee J) \vee (0 = 0)))$$

Отметим, что вплоть до данного этапа сохраняется эквивалентность полученной формулы и условия корректности.

Далее на шаге 5 был выполнен шаг 2 преобразования. Применение шага 2 на текущем этапе исполнения стратегии привело к преобразованию в формуле ϕ_5 только следующего дизъюнкта $(M \Rightarrow ((K \vee \neg J) \vee (1 = 0)))$. Поэтому, рассмотрим применение стратегии, соответствующей шагу 2, именно к этому дизъюнкту. Обозначим его как S. Для него был построен граф соотношений: Рассмотрим следующую связную многовершинную компоненту этого графа:

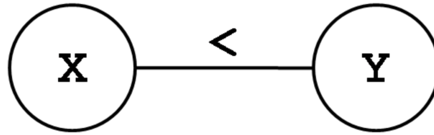

Рис. 5.3. Компонента графа соотношений

Рисунок 5.3 представляет интересующую нас компоненту графа соотношений. Рассмотрим пометки вершин. Пометка X обозначает $(cnt\ 0\ (-\ n\ 1)\ key\ a)$, а пометка Y обозначает $entr$. Отметим, что эта компонента является формулой E.

К преобразованиям формулы ϕ_5 привело рассмотрение на текущем шаге цели $\neg J$. Она соответствует шаблону $g(c, d)$, где отношение "g" – это $\neq$", при этом "c" – это переменная $entr$ и "d" – это выражение $(rep1\ n\ key\ entr\ a)$. Для данного отношения была запущена специальная процедура поиска соответствующего переменной $entr$ вызова функции. Поэтому, поиск начался от вершины с пометкой X. В процессе поиска был построен подграф графа соотношений. Им является приведенная на рисунке 5.3 компонента графа соотношений. Выражение $(cnt\ 0\ (-\ n\ 1)\ key\ a)$ является искомым вызовом функции. Конъюнкт E является искомым путем. Выражение $(cnt\ 0\ (-\ n\ 1)\ key\ a)$ стало значением переменной v, а конъюнкт E стал значением переменной q.

Таким образом, был порождена новая формула

$$(=\ (cnt\ 0\ (-\ n\ 1)\ key\ a)\ (rep1\ n\ key\ entr\ a)).$$

Обозначим ее как T.

Пусть Z обозначает формулу, которая была получена из дизъюнкта S путем замены цели $\neg J$ на цель T. Рассмотрим формулу Z:

$$(M \Rightarrow ((K \vee T) \vee (1 = 0))).$$

Отметим, что формулы Z и S не эквивалентны, но из истинности формулы Z следует истинность формулы S. Поэтому, формула ϕ_6 получена из формулы ϕ_5 путем замены конъюнкта S на конъюнкт Z. Таким образом, формула ϕ_6 имеет следующий вид:

$$(L \Rightarrow (\neg K \vee (0 = 1))) \wedge (L \Rightarrow ((K \vee \neg J) \vee (1 = 1))) \wedge (L \Rightarrow ((K \vee J) \vee (0 = 1))) \wedge$$
$$(M \Rightarrow (\neg K \vee (0 = 0))) \wedge (M \Rightarrow ((K \vee T) \vee (1 = 0))) \wedge (M \Rightarrow ((K \vee J) \vee (0 = 0)))$$

Отметим, что на текущем этапе исполнения стратегии эквивалентность между полученной формулой и условием корректности не сохраняется, но полученная формула сильнее исходного условия корректности (то есть из истинности полученной формулы следует истинность условия корректности).

На следующем шаге 6 истинность полученной формулы ϕ_6 была доказана в системе ACL2 индукцией по "n". Таким образом, результатом работы алгоритма является утверждение "формула ϕ истинна". Отметим, что наша стратегия доказательства позволила проверить условие корректности на истинность в автоматическом режиме.

5.5.2. C-представление Sisal программы, проверяющей упорядоченность элементов массива по возрастанию

Рассмотрим верификацию Sisal-программы, проверяющую, упорядочен ли массив в диапазоне $[0,\ n-1]$, с помощью модуля CSV1 [123]:

```
function is_ordered (a: array of integer, n, integer returns integer)
    let count :=
```

```
        for cnt := 0, i in 0..n-2
        while !(a[i] <= a[i+1]) do
            cnt := cnt + 1;
        returns value of cnt end for
    in result := if count=n then 1 else 0
    end let end function
```

Для этой программы было сгенерировано следующее C-light представление:

```
int is_ordered(int* a, int n){
   int count = 1, result = 0, i = 2;
   for (i = 0; i < n; i++){
       if (a[i-2] < a[i-1]) count = count + 1;
       if (count == n) {result = 1; break;}}
   return result;}
```

Рассмотрим полученное системой C-lightVer условие корректности C-light представления данной программы. Обозначим данное условие корректности как ϕ. Формула ϕ имеет следующий вид [4]:

$$\forall n \in Int, a \in IntArr$$
$$n = f(2,\ n,\ a) \rightarrow 1 = g(n,\ a),$$

где Int – множество целых чисел и $IntArr$ — множество целочисленных массивов, функция f определена на языке Lisp следующим образом:

```
(defun f(low, m, arr)
    (if (< k low) 1
        (if (<= (nth (- m 2) arr) (nth (- m 1) arr))
            (+ 1 (f (- m 1) arr)) (h (- m 1) arr))))
```

Функция nth реализует индексируемый доступ к элементу массива. Функция h считает количество элементов, не меньших предыдущего, в диапазоне массива $[low - 1,\ k]$. Рассмотрим определение функции g на языке системы ACL2 :

```
(defun g(k, arr)
    (if (= k (h k arr)) 1 0))
```

где функция h определена на языке системы ACL2 следующим образом:

```
(defun h(l, arr)
    (if (<= l 1) 1
        (if (<= (nth (- l 2) arr) (nth (- l 1) arr))
            (+ 1 (h (- l 1) arr)) (h (- l 1) arr))))
```

Функция h считает количество элементов, не меньших предыдущего, в диапазоне массива $[0,\ k]$. Отметим, что функции g и h реализуют операцию замены. Рекурсивное определение функций задано на языке системы ACL2. Но система ACL2 не смогла доказать исходную формулу индукцией по n, поэтому, была применена стратегия усиления УК, описанная в разделе 2.4.2.

Шаги (1)-(3) стратегии не изменили формулу ϕ. Так как функция g определена нерекурсивно, то выражение $1 = g(n,\ a)$ заменяется следующим:

$$n = h(n,\ a) \to 1 = 1 \land \neg(n = h(n,\ a)) \to 1 = 0$$

Это выражение получено с помощью преобразования обоих ветвей выражения if из функции g в две импликации соответственно. Таким образом, после исполнения шага (4) была получена следующая формула:

$$\begin{gathered} \forall n \in Int, a \in IntArr \\ n = f(2,\ n,\ a) \to \\ (n = h(n,\ a) \to 1 = 1 \land \neg(n = h(n,\ a)) \to 1 = 0) \end{gathered}$$

Отметим, что эквивалентность исходной и полученной формулы на текущем этапе сохраняется. Но, структура исходной формулы была изменена, поэтому, на шаге (5) исполнение шагов (1)-(3) стратегии запускается снова.

На шаге (1) стратегии формула была преобразована следующим образом:

$$\begin{gathered} \forall n \in Int, a \in IntArr \\ (n = f(2,\ n,\ a) \to n \neq h(n,\ a) \lor 1 = 1) \land (n = f(2,\ n,\ a) \to n = h(n,\ a) \lor 1 = 0) \end{gathered}$$

Отметим, что стратегия не предлагает вычисление константных выражений, таких как $1 = 1$, так как система ACL2 поддерживает вычисление такие выражений. Обозначим первый конъюнкт как ψ:

$$\forall n \in Int, a \in IntArr$$
$$n = f(2,\ n,\ a) \rightarrow n \neq h(n,\ a) \vee 1 = 1$$

Так как система ACL2 вычисляет истинность выражения $1 = 1$, ψ доказывается в автоматическом режиме. Поэтому, рассмотрим второй конъюнкт. Обозначим его как ω:

$$\forall n \in Int, a \in IntArr$$
$$n = f(2,\ n,\ a) \rightarrow n = h(n,\ a) \vee 1 = 0$$

Отметим, что система ACL2 не смогла доказать данную формулу индукцией по n в автоматическом режиме. Трудностью для системы ACL2 является равенство переменной и применения функции. На шаге (2) стратегии рассмотрим цель $n = h(n,\ a)$. Для посылки формулы был построен граф соотношений, состоящий из двух вершин, помеченных как n и $f(2,\ n,\ a)$ соответственно. Соединяющее эти вершины ребро помечено как $=$. Поэтому, в результате применения шага (2) стратегии была получена следующая формула:

$$\forall n \in Int, a \in IntArr$$
$$f(2,\ n,\ a) = h(n,\ a) \vee 1 = 0$$

Данная формула была успешно доказана в системе ACL2 индукцией по n в автоматическом режиме. Так как стратегия усиления УК корректна, то исходная формула ϕ истинна.

5.5.3. Время исполнения реализации модуля CSV1

Так как корректность стратегии усиления УК, доказана, то программы, верифицированные в наших экспериментах, корректны. Теперь следует проверить, допустимое ли время экспериментов получается в ходе практического применения данных подходов.

Эксперименты были проведены с помощью виртуальной машины Oracle VM VirtualBox (процессор Intel Core i7-3520M, 2.9 Ггц, 4 Гб оперативной памяти) [4]. Продолжительность сеанса верификации складывается из продолжительности трансляции в промежуточное представление, продолжительности генерации условий корректности и продолжительности доказательства условия корректности в системе ACL2.

Приведем таблицу 5.1 с продолжительностью сеансов верификации двух приведенных ранее Cloud Sisal программ.

Пример	search_count	is_ordered
Продолжительность сеанса верификации	65 сек	53 сек

Таблица 5.1. Эксперименты по верификации Cloud Sisal программ

В результате применения описанной стратегии продолжительность сеансов верификации является приемлемой [123].

5.5.4. Сумма элементов матрицы на языке Cloud-Sisal-kernel

Рассмотрим верификацию программы суммирования элементов матрицы с помощью модуля CSV2 [130]:

```
function sum_matrix_elements (a: array of (array of integer),
                    n, m: integer returns integer)
 for i in 0..n-1..1 cross j in 0..m-1..1 do
 returns sum of a[i, j] end for end function
```

Данная программа принимает в качестве аргументов матрицу a с n строками и m столбцами.

Предусловием является следующая формула на языке системы ACL2:

```
(and (integerp n) (integerp m) (< 0 n) (< 0 m) (integer-matrixp n m a))
```

Предикат *integerp* проверяет, является ли аргумент целым числом. Предикат *integer-matrixp* проверяет, является ли аргумент целочисленной матрицей. В

теории предметной области, заданной в системе ACL2, целочисленная матрица моделируется списком n подсписков, каждый из которых является целочисленным списком длины m.

Предусловием является следующая формула на языке системы ACL2:

```
(= result (sum-matrix n m a))
```

где *sum-matrix* является функцией из теории предметной области, реализующей сумму элементов матрицы.

Циклическое выражение транслируется в следующее применение функции rep_1:

```
(rep_1 (reverse (cartesian_product (triplet 0 (- n 1) 1) (triplet 0 (- m 1) 1)))
       (create_environment_1 a))
```

Были сгенерированы следующие определения rep_1 и $create_environment_1$:

```
(defun create_environment_1(a) (make-environment_1 :a a))
(defun rep_1(range_tuples environment)
  (b* ((when (endp range_tuples) 0) (tuple (car range_tuples))
       (i (car tuple)) (j (car (cdr tuple))) (a environment.a))
    (+ (nth i (nth j a)) (rep_1 (cdr range_tuples) environment))))
```

Переменные диапазона цикла i и j не входят в контекст, поэтому контекст редуцируемого выражения $a[i, j]$ состоит только из переменной a. Отметим, что функция rep_1 задает способ исполнения цикла, некоторую операционную семантику цикла.

Чтобы вывести слабейшее предусловие для функции *sum-matrix-elements* и ее постусловия, каждое вхождение терма *result* в постусловии заменяется на результат трансляции циклического выражения, составляющего тела цикла. В результате такой замены получается следующее слабейшее предусловие:

```
(= (rep_1 (reverse (cartesian_product (triplet 0 (- n 1) 1)
                                      (triplet 0 (- m 1) 1)))
          (create_environment_1 a)) (sum-matrix n m a))
```

Функция *sum-matrix-elements* частично корректна относительно ее аннотаций, если истинно следующее УК:

```
(implies (and (integerp n) (integerp m) (< 0 n) (< 0 m) (integer-matrixp n m a))
 (= (rep_1 (reverse (cartesian_product (triplet 0 (- n 1) 1)
                                       (triplet 0 (- m 1) 1)))
          (create_environment_1 a)) (sum-matrix n m a)))
```

Посылкой данной импликации является предусловие функции *sum-matrix-elements*. Слабейшее предусловие для *sum-matrix-elements* и ее постусловия составляет заключение данной импликации. Система ACL2 автоматически доказывает данное УК, используя индукцию по n и m, а также теорию предметной области о целочисленных матрицах. Таким образом, *sum-matrix-elements* соответствует своим спецификациям.

5.5.5. Произведение элементов матрицы на языке C-Sisal-kernel

Рассмотрим автоматическую верификацию с помощью C-lightVer и CSV2 программы, заданной на языке C-Sisal-kernel. Эта программа вычисляет произведение элементов целочисленной матрицы [15]:

```
int product_matrix_elements (int** a, int n, int m){
  loop_expr prod = for i in 0..n-1..1 cross j in 0..m-1..1 do
                   returns product of a[i][j] end for;
  return prod.scalar;}
```

Входными данными этой функции является целочисленная матрица a с n строками и m столбцами. Рассмотрим предусловие этой функции, заданное на языке ACL:

```
(and (integerp n) (integerp m) (< 0 n) (< 0 m) (integer-matrixp n m a))
```

Предикат *integerp* проверяет, является ли значение его аргумента целым числом. Целочисленная матрица задана как список целочисленных списков. Длины всех вложенных списков равны. Предикат *integer-matrixp* проверяет, соответствует ли его аргумент этой структуре.

Постусловием является равенство `(= prod (product-matrix n m a))`. Функция *product-matrix* задана в теории предметной области. Циклическое выражение транслируется в следующую структуру, основанную на применении функции rep_1:

```
(make-loop_expr
  :scalar (rep_1
   (reverse (cartesian_product (triplet 0 (- n 1) 1) (triplet 0 (- m 1) 1)))
   (create_environment_1 a))
  :vector nil).scalar
```

Рассмотрим определение функции $create_environment_1$ и определение функции rep_1:

```
(defun create_environment_1(a) (make-environment_1 :a a))
(defun rep_1 (range_tuples environment)
  (b* (((when (endp range_tuples)) 1) (tuple (car range_tuples))
       (i (car tuple)) (j (car (cdr tuple))) (a environment.a)
       (previous_iter_result (rep_1 (cdr range_tuples) environment)))
    (* (nth j (nth i a)) previous_iter_result)))
```

Метод смешанной аксиоматической семантики [147] и метод метагенерации УК [124] позволили применить правило вывода без MeM и MD. Полученное слабейшее предусловие основано на замене переменной *prod* в постусловии на результат трансляции цикла. Рассмотрим результат этой замены:

```
(= (make-loop_expr
   :scalar (rep_1
    (reverse (cartesian_product (triplet 0 (- n 1) 1) (triplet 0 (- m 1) 1)))
    (create_environment_1 a))
   :vector nil ).scalar
   (product-matrix n m a))
```

Рассмотрим полученное УК:

```
(implies (and (integerp n) (integerp m) (< 0 n) (< 0 m) (integer-matrixp n m a))
  (= (make-loop_expr
```

```
    :scalar (rep_1
     (reverse (cartesian_product (triplet 0 (- n 1) 1) (triplet 0 (- m 1) 1)))
     (create_environment_1 a))
    :vector nil
  ).scalar
  (product-matrix n m a)))
```

Система ACL2 автоматически доказала это УК, используя индукцию по n и m. Это доказательство основано на использовании лемм о *integer-matrixp* и *product-matrix*. Следовательно, функция `product_matrix_elements` соответствует спецификациям.

5.5.6. Программа на языке C-Sisal-kernel, изменяющая знак первого отрицательного элемента массива на противоположный

Рассмотрим автоматическую верификацию с помощью C-lightVer и CSV2 программы `negate_first` из набора задач по верификации [100]. Эта программа изменяет знак первого отрицательного элемента массива на противоположный. Главной проблемой верификации программы `negate_first` является использование в теле цикла конструкции, подобной конструкции `break`. Рассмотрим функцию `negate_first`, записанную на языке C-Sisal-kernel:

```
void negate_first(int n, int* a){
    loop_expr index = for i in 0..n-1..1 while a[i] >= 0 do
                      returns value of i end for;
    if (a[index.scalar] < 0) {a[index.scalar] = -a[index.scalar];}}
```

Предусловие этой функции задано на языке ACL:

```
(and (integer-listp a) (integer-listp a_0) (equal a a_0)
     (integerp n) (< 0 n) (<= n (length a_0)))
```

Рассмотрим постусловие этой функции, заданное на языке ACL:

```
(and (implies (not (found-negative n a_0)) (equal a a_0))
     (implies (found-negative n a_0)
              (equal a (update-nth (count-index n a_0)
                                   (- (nth (count-index n a_0) a_0)) a_0)))))
```

В постусловии используется предикат *found-negative*, заданный нами. Этот предикат проверяет наличие отрицательного элемента в массиве. Также в постусловии используется функция *count-index*, заданная нами. Эта функция вычисляет индекс первого отрицательного элемента массива в случае наличия в массиве такого элемента. Значение этой функции не определено в других случаях.

Данное постусловие является конъюнкцией импликаций. Первая импликация соответствует случаю отсутствия отрицательных элементов в массиве. Полученный массив совпадает с исходным массивом в данном случае. Вторая импликация соответствует случаю наличия отрицательного элемента в массиве. В этом случае полученный массив соответствует исходному за исключением первого отрицательного элемента.

Циклическое выражение транслируется в применение функции rep_1 к применениям функций *partition* и $create_environment_1$. Рассмотрим определение функции $create_environment_1$ и определение функции rep_1:

```
(defun create_environment_1(a) (make-environment_1 :a a))
(defun rep_1 (range_tuples environment)
  (b* (((when (endp range_tuples)) (update :loop-break nil 0))
       (tuple (car range_tuples))
       (i (car tuple)) (a environment.a)
       (previous_iter_result (rep_1 (cdr range_tuples) environment))
       ((when previous_iter_result.loop-break) previous_iter_result)
       ((when (not (>= (nth i a) 0))) (update :loop-break t
                                             previous_iter_result))) i ))
```

где поле *loop-break* является истинным тогда и только тогда, когда конструкция `while` завершает исполнение циклического выражения.

Метод смешанной аксиоматической семантики [147] и метод метагенерации УК [124] позволили применить правило вывода без MeM и MD. Наличие инструкции if привело к генерации двух УК. Рассмотрим одно из них:

```
(implies (and (integer-listp a) (integer-listp a_0) (equal a a_0)
              (integerp n) (< 0 n) (<= n (length a_0))
```

```
    (>= (nth (make-loop_expr :scalar (rep_1
                               (reverse (partition (triplet 0 (- n 1) 1)))
                               (create_environment_1 a))
                              :vector nil).scalar a) 0)
(and (implies (not (found-negative n a_0))
  (equal (update-nth
           (make-loop_expr :scalar (rep_1
                             (reverse (partition (triplet 0 (- n 1) 1)))
                             (create_environment_1 a))
                            :vector nil).scalar
    (- (nth (make-loop_expr :scalar (rep_1
                             (reverse (partition (triplet 0 (- n 1) 1)))
                             (create_environment_1 a))
                            :vector nil).scalar a)) a) a_0))
 (implies (found-negative n a_0)
  (equal (update-nth
    (make-loop_expr :scalar (rep_1
                      (reverse (partition (triplet 0 (- n 1) 1)))
                      (create_environment_1 a))
                     :vector nil).scalar
    (- (nth (make-loop_expr :scalar (rep_1
                             (reverse (partition (triplet 0 (- n 1) 1)))
                             (create_environment_1 a))
                            :vector nil).scalar a)) a)
   (update-nth (count-index n a_0)
               (- (nth (count-index n a_0) a_0)) a_0))))) )
```

Второе УК схоже с рассмотренным. Структура УК соответствует импликации из предусловия в постусловие с заменой вхождений a на выражения *update-nth*, где вместо индекса используется значение функции rep_1.

Стратегия для программ, постусловием которых является разбор случаев выхода из цикла, описанная в разделе 2.4.4, позволила системе ACL2 доказать леммы о том, что первая импликация в постусловии эквивалентна случаю отсутствия завершения цикла конструкцией `while` и вторая импликация в постусловии эквивалентна случаю завершения цикла конструкцией `while` [15]. Эти леммы были добавлены в теорию предметной области. Система ACL2 автомати-

чески доказала оба УК, используя индукцию по n и эти леммы. Следовательно, функция `negate_first` соответствует спецификациям.

5.6. Выводы

На основе сделанного описания применения комплексного подхода к дедуктивной верификации программ на языках Cloud Sisal и C-Sisal-kernel можно сделать следующие выводы:

- Разработанная аксиоматическая семантика языка Cloud Sisal и языка C-Sisal-kernel позволяет верифицировать программы с финитными итерациями на данных языках без инвариантов циклов. Применение символического метода верификации финитных итераций не только к C-программам, но и к программам на языках Cloud Sisal и C-Sisal-kernel демонстрирует универсальность данного метода.

- Отметим схожесть алгоритмов генерации функций, выражающих результат финитных итераций, заданных и на языке C, и на языке Cloud Sisal, и на языке C-Sisal-kernel. Это демонстрирует универсальность данного алгоритма, которая может упростить задание такого алгоритма для финитных итераций на других языках.

- Стратегии доказательства УК позволяют автоматизировать доказательство не только УК C-программ, но и УК программ на языках Cloud Sisal и C-Sisal-kernel. Это демонстрирует универсальность стратегий, основанных на генерации свойств об операции замены rep.

- Продемонстрирована универсальность комплексного подхода к автоматизации дедуктивной верификации программ с финитными итерациями и возможность его применения к широкому классу языков императивного и функционального программирования, позволяющих задавать финитные итерации над структурами данных.

Заключение

Основные научные и практические результаты, полученные в диссертационной работе и выносимые на защиту, состоят в следующем:

1. Разработанный метод генерации операции замены для циклов позволяет генерировать условия корректности для программ с финитными итерациями без использования инвариантов циклов. Данный метод включает в себя следующие алгоритмы: генерации функций, выражающих результаты итераций с вложенными условными инструкциями над неизменяемыми массивами и генерации функций, выражающих результаты итераций над изменяемыми массивами.

2. Разработанные стратегии доказательства условий корректности позволяют автоматизировать проверку на истинность условий корректности программ с финитными итерациями. Были предложены следующие стратегии: выбора посылок, для программ, постусловием которых является разбор случаев выхода из цикла, для программ с финитными итерациями над массивами, интерактивного доказательства, усиления условий корректности, для программ, спецификации которых содержат функции со свойством конкатенации, для финитных итераций с инструкцией `break`, для программ с выходом из цикла, для финитных итераций над изменяемыми массивами, для программ с вложенными циклами. Доказана корректность стратегии усиления условий корректности.

3. Предложенный метод локализации ошибок позволяет автоматизировать сопоставление конструкций программы и подформул условий корректности, а также локализацию ошибок в программах с финитными итерациями. Были разработаны язык представления семантических меток, семантические метки для функций, выражающих результаты финитных итераций, алгоритм генерации функций, выражающих результаты финитных итераций, с семантическими метками для итераций над изменяемыми массивами, алгоритм генерации объяснений недоказанных условий коррект-

ности, содержащих операцию замены, стратегия проверки ложности недоказанных условий корректности, стратегия поиска циклов с неиспользуемыми присваиваниями элементам массива, стратегия проверки исполнения инструкции `break` на первой итерации цикла.

4. Разработанная аксиоматическая семантика языка Cloud-Sisal-kernel позволяет при дедуктивной верификации программ на данном языке использовать правила вывода для циклических выражений без инвариантов. Предложенная аксиоматическая семантика расширения (C-Sisal-kernel) языка C конструкциями языка Sisal позволяет применять в системе CPPS методы комплексного подхода системы C-lightVer. Разработан алгоритм генерации функций, выражающих результат циклических выражений языка Cloud-Sisal-kernel и языка C-Sisal-kernel.

5. Реализация методов комплексного подхода в системе C-lightVer, позволяющая при дедуктивной верификации программ с финитными итерациями, заданных на языках C, Cloud Sisal и на расширении C конструкциями Cloud Sisal, автоматизировать доказательство условий корректности и автоматизировать локализацию ошибок. Были проведены эксперименты по автоматизированной верификации C-представлений Sisal программ, программ на языке Cloud-Sisal-kernel, программ на языке C-Sisal-kernel. Были проведены эксперименты по автоматизированной верификации C-программ, представляющих следующие классы: программы с финитными итерациями над неизменяемыми массивами без инструкции `break`, программы с финитными итерациями над неизменяемыми массивами с инструкциями `break`, программы с финитными итерациями над изменяемыми массивами с инструкциями `break`. Проведен успешный эксперимент по автоматической верификации программы сортировки простыми вставками без инвариантов циклов.

Перспективы развития направления исследований. В будущем планируется разработать расширение комплексного подхода на программы с более

общими видами итераций над различными структурами данных. Примерами таких структур данных являются файлы, списки, деревья. В качестве примера итераций над ними можно рассмотреть реализации классических алгоритмов над такими структурами данных, в том числе различные реализации топологической сортировки ациклического ориентированного графа. Расширение комплексного подхода может включать новые алгоритмы генерации операций замены, новые стратегии автоматизации доказательства УК и новые методы автоматизации локализации ошибок. Представленные в диссертации методы комплексного подхода могут упростить разработку и реализацию такого расширения.

Список литературы

1. Видеоканал ИСИ СО РАН [Электронный ресурс] // Режим доступа: https://www.youtube.com/channel/UCvMEarh2NIJHZykxL9iJ9Ww, свободный (дата обращения: 30.03.2022).
2. Глуханков М. П., Дортман П. А., Павлов А. А., Стасенко А. П. Транслирующие компоненты системы функционального программирования SFP // Современные проблемы конструирования программ. Сер. «Конструирование и оптимизация программ». — Новосибирск, 2002. — С. 69–87.
3. Камкин А.С. Введение в формальные методы верификации программ: Учебное пособие. М.: МАКС Пресс, 2018. — 272 с.
4. Касьянов В. Н., Гордеев Д. С., Золотухин Т. А., Касьянова Е. В., Кондратьев Д. А. Система облачного параллельного программирования CPPS: визуализация и верификация Cloud Sisal программ. Новосибирск: ИПЦ НГУ, 2020. — 256 с.
5. Касьянов В. Н., Касьянова Е. В. Язык программирования Cloud Sisal. — Новосибирск, 2018. — 55 с. — (Препр. / ИСИ СО РАН; № 181).
6. Касьянов В. Н., Стасенко А. П. Язык программирования Sisal 3.2 // Методы и инструменты конструирования программ. Сер. «Конструирование и оптимизация программ». — Новосибирск, 2007. — С. 56–134.
7. КолТЭП-2019. Утренняя сессия. Часть 2 [Электронный ресурс] // Режим доступа: https://www.youtube.com/watch?v=W-Fq8b_JH6M, свободный (дата обращения: 31.03.2022).
8. Кондратьев Д. А. Анализ аннотированных Си-программ и их трансляция в промежуточное представление: выпускная квалификационная работа бакалавра по направлению подготовки 230100.62 «Информатика и вычислительная техника». — Новосибирск, 2013. — 45 с.
9. Кондратьев Д. А. Анализ аннотированных Си-программ и их трансляция в промежуточное представление // Материалы 51 Международной научной студенческой конф. «Студент и научно-технический прогресс»: Информационные технологии, Новосибирск, Россия, 12–18 апреля, 2013. —Новоси-

бирск: Изд–во Новосиб. гос. ун–та, 2013. — С. 202.

10. Кондратьев Д. А. Верификация программ инженерной математики в системе C-light // Материалы XVIII Всероссийской конф. молодых ученых по математическому моделированию, Иркутск, Россия, 21 – 25 августа, 2017. — Новосибирск: Институт вычислительных технологий СО РАН, 2017. — С. 78.
11. Кондратьев Д. А. Доказательство условий корректности Си-программ, осуществляющих финитные итерации над структурами данных // Тез. XIX Всероссийской конф. молодых ученых по математическому моделированию и информационным технологиям, Кемерово, Россия, 29 октября – 2 ноября, 2018. – Новосибирск: Институт вычислительных технологий СО РАН, 2018. — С. 65.
12. Кондратьев Д. А. Метагенератор условий корректности для языка Си // Материалы 52 Международной научной студенческой конф. «Студент и научно-технический прогресс»: Информационные технологии, Новосибирск, Россия, 11–18 апреля, 2014. — Новосибирск: Изд–во Новосиб. гос. ун–та, 2014. — С. 126.
13. Кондратьев Д. А. Модификации метода автоматизации локализации ошибок в C-программах, реализованного в системе C-lightVer // Тез. XXII Всероссийской конф. молодых ученых по математическому моделированию и информационным технологиям, Новосибирск, Россия, 25 – 29 октября, 2021. — Новосибирск: ФИЦ ИВТ, 2021. — С. 54.
14. Кондратьев Д. А. На пути к автоматической верификации C-программ с вложенными циклами в системе C-lightVer // Тез. XX Всероссийской конф. молодых ученых по математическому моделированию и информационным технологиям, Новосибирск, Россия, 28 октября – 1 ноября, 2019. — Новосибирск: Институт вычислительных технологий СО РАН, 2019. — С. 62.
15. Кондратьев Д.А. На пути к автоматической дедуктивной верификации C-программ с Sisal-циклами в системе C-lightVer // Моделирование и анализ информационных систем. — 2021. — Т. 28, № 4. — С. 372–393.

16. Кондратьев Д. А. На пути к дедуктивной верификации C-программ, расширенных циклами языка Cloud Sisal // Тез. XXI Всероссийской конференции молодых ученых по математическому моделированию и информационным технологиям, Новосибирск, Россия, 7 – 11 декабря, 2020. —Новосибирск: ФИЦ ИВТ, 2020. — С. 35–36.
17. Кондратьев Д. А. Программа-транслятор аннотированных C-программ в семантически аксиоматизируемое представление для задач верификации. Свидетельство о государственной регистрации программы для ЭВМ № 2021614956, 01.04.2021.
18. Кондратьев Д. А. Расширение метагенерации условий корректности концепцией семантической разметки // Материалы третьей международной научно-практической конф. «Инструменты и методы анализа программ», Санкт-Петербург, 12–14 ноября, 2015. — СПб.: Изд–во С.-Петерб. Политехн. ун–та Петра Великого, 2015. — С. 107–118.
19. Кондратьев Д. А. Расширение системы C-light символическим методом верификации финитных итераций // Вычислительные технологии. — 2017. — Т. 22, Специальный выпуск 1. — С. 44–59.
20. Кондратьев Д. А. Расширение системы C-light символическим методом верификации финитных итераций // Материалы XVII Всероссийской конф. молодых ученых по математическому моделированию, Новосибирск, Россия, 30 октября – 3 ноября, 2016. — Новосибирск: Институт вычислительных технологий СО РАН, 2016. — С. 91–92.
21. Кондратьев Д.А. Расширяемый генератор условий корректности для языка Си: магистерская диссертация по направлению подготовки 09.04.01 «Информатика и вычислительная техника». — Новосибирск, 2015. — 50 с.
22. Кондратьев Д. А. Самоприменимый метагенератор условий корректности для языка Си // Материалы 53 Международной научной студенческой конф. «Студент и научно-технический прогресс»: Информационные технологии, Новосибирск, Россия, 11–17 апреля, 2015. — Новосибирск: Изд–во Новосиб. гос. ун–та, 2015. — С. 136.
23. Кондратьев Д. А. Семантические метки в проекте C-light // Материалы

XVI Всероссийской конф. молодых ученых по математическому моделированию, Красноярск, Россия, 28–30 октября 2015. — Новосибирск: Институт вычислительных технологий СО РАН, 2015. — С. 76.

24. Кондратьев Д. А., Промский А. В. Аксиоматическая семантика подмножества языка Cloud Sisal для верификации программ инженерной математики // Тез. Междунар. конф. «Марчуковские научные чтения 2020», посв. 95-летию со дня рождения акад. Г. И. Марчука, Новосибирск, Россия, 19 – 23 октября 2020. — Новосибирск : ИПЦ НГУ, 2020. — С. 106.
25. Кондратьев Д. А., Промский А. В. Комплексный подход к локализации ошибок при верификации Си-программ // Системная информатика. — 2013, № 1. — С. 79–96.
26. Марьясов И. В. Верификация C-программ с помощью смешанной аксиоматической семантики: дис... канд. физ.-мат. наук: 05.13.11. — Новосибирск, 2012. — 110 с.
27. Миронов А. М. Верификация программ. Часть 1: нерекурсивные программы. М.: МАКС Пресс, 2017. — 76 с.
28. Непомнящий В. А., Ануреев И. С., Михайлов И. Н., Промский А. В. На пути к верификации C-программ. Часть 1. Язык C-light. — Новосибирск, 2001. — 48 с. — (Препр. / ИСИ СО РАН; № 84).
29. Непомнящий В. А., Ануреев И. С., Михайлов И. Н., Промский А. В. На пути к верификации C-программ. Часть 2. Язык C-light-kernel и его аксиоматическая семантика. — Новосибирск, 2001. — 57 с. — (Препр. / ИСИ СО РАН; № 87).
30. Непомнящий В. А., Ануреев И. С., Михайлов И. Н., Промский А. В. На пути к верификации C программ. Часть 3. Перевод из языка C-light в язык C-light-kernel и его формальное обоснование. — Новосибирск, 2002. — 82 с. — (Препр. / ИСИ СО РАН; № 97).
31. Непомнящий В. А., Ануреев И. С., Михайлов И. Н., Промский А. В. Ориентированный на верификацию язык C-light // Системная информатика. — Новосибирск, 2004. — Вып. 9. — С. 51–134.
32. Непомнящий В. А., Рякин О. М. Прикладные методы верификации про-

грамм. М.: Радио и связь, 1988. — 256 с.

33. Промский А. В. Формальная семантика C-light программ и их верификация методом Хоара: дис... канд. физ.-мат. наук: 05.13.11. — Новосибирск, 2004. — 157 с.
34. Alkassar E., Hillebrand M.A., Leinenbach D., Schirmer N. W., Starostin A. The Verisoft Approach to Systems Verification // Lect. Notes Comput. Sci. — Berlin etc., 2008. — Vol. 5295. — P. 209–224.
35. Ameri M., Furia C. A. Why Just Boogie? // Lect. Notes Comput. Sci. — Cham, 2016. — Vol. 9681. — P. 79–95.
36. Appel A. W., Blazy S. Separation Logic for Small-Step cminor // Lect. Notes Comput. Sci. — Berlin etc., 2007. — Vol. 4732. — P. 5–21.
37. Apt K. R., Olderog E.-R. Assessing the Success and Impact of Hoare's Logic // Theories of Programming: The Life and Works of Tony Hoare. — New York: ACM, 2021. — P. 41–76.
38. Apt K. R., Olderog E.-R. Fifty years of Hoare's logic // Formal Aspects of Computing. — 2019. — Vol. 31, Issue 6. — P. 751–807.
39. Attali I., Caromel D., Wendelborn A. A Formal Semantics and an Interactive Environment for Sisal // The Springer International Series in Software Engineering. — Boston, 1996. — Vol. 2. — P. 229–256.
40. Barnett M., Chang B.-Y. E., DeLine R., Jacobs B., Leino K. R. M. Boogie: A Modular Reusable Verifier for Object-Oriented Programs // Lect. Notes Comput. Sci. — Berlin etc., 2006. — Vol. 4111. — P. 364–387.
41. Barnett M., Fähndrich M., Leino K.R.M., Müller P., Schulte W., Venter H. Specification and verification: the Spec# experience // Communications of the ACM. — 2011. — Vol. 54, Issue 6. — P. 81–91.
42. Barnett M., Leino K. R. M., Schulte W. The Spec# Programming System: An Overview // Lect. Notes Comput. Sci. — Berlin etc., 2005. — Vol. 3362. — P. 49—69.
43. Barrett C., Conway C. L., Deters M., Hadarean L., Jovanović D., King T., Reynolds A., Tinelli C. CVC4 // Lect. Notes Comput. Sci. — Berlin etc., 2011. — Vol. 6806. — P. 171–177.

44. Bartocci E., Beyer D., Black P. E., Fedyukovich G., Garavel H., Hartmanns A., Huisman M., Kordon F., Nagele G., Sighireanu M., Steffen B., Suda M., Sutcliffe G., Weber T., Yamada A. TOOLympics 2019: An Overview of Competitions in Formal Methods // Lect. Notes Comput. Sci. — Cham, 2019. — Vol. 11429. — P. 3–24.
45. Baudin P., Bobot F., Bühler D., Correnson L., Kirchner F., Kosmatov N., Maroneze A., Perrelle V., Prevosto V., Signoles J., Williams N. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform // Communications of the ACM. — 2021. — Vol. 64, Issue 8. — P. 56–68.
46. Baudin P., Cuoq P., Filliâtre J. C., Marché C., Monate B., Moy Y., Prevosto V. ACSL: ANSI/ISO C Specification Language [Электронный ресурс] // Режим доступа: https://frama-c.com/download/acsl-1.17.pdf, свободный (дата обращения: 17.03.2022).
47. Becker B., Lourenço C.B., Marché C. Explaining Counterexamples with Giant-Step Assertion Checking // Proc. 6th Workshop on Formal Integrated Development Environment, May 24-25, 2021. — Electronic Proc. Theor. Comput. Sci. — 2021. — Vol. 338. — P. 82–88.
48. Beckert B., Klebanov V., Weiß B. Dynamic Logic for Java // Lect. Notes Comput. Sci. — Cham, 2016. — Vol. 10001. — P. 49–106.
49. Besson F., Blazy S., Wilke P. CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics // Journal of Automated Reasoning. — 2019. — Vol. 63, Issue 2. — P. 369–392.
50. Beyer D. Advances in Automatic Software Verification: SV-COMP 2020 // Lect. Notes Comput. Sci. — Cham, 2020. — Vol. 12079. — P. 347–367.
51. Bjørner N., Nachmanson L. Navigating the Universe of Z3 Theory Solvers // Lect. Notes Comput. Sci. — Cham, 2020. — Vol. 12475. — P. 8–24.
52. Blanc R., Kuncak V., Kneuss E., Suter P. An overview of the Leon verification system: verification by translation to recursive functions // Proc. 4th Workshop on Scala, Montpellier, France, July 2, 2013. — New York: ACM, 2013. — Article ID: 1. — P. 1–10.
53. Blanchard A., Loulergue F., Kosmatov N. Towards Full Proof Automation in

Frama-C Using Auto-active Verification // Lect. Notes Comput. Sci. — Cham, 2019. — Vol. 11460. — P. 88–105.

54. Blazy S., Leroy X. Mechanized Semantics for the Clight Subset of the C Language // Journal of Automated Reasoning. — 2009. — Vol. 43, Issue 3. — P. 263–288.
55. Blom S., Darabi S., Huisman M., Safari M. Correct program parallelisations // International Journal on Software Tools for Technology Transfer. — 2021. — Vol. 23, Issue 5. — P. 741–763.
56. Bobot F., Filliâtre J. C., Marché C., Paskevich A. Let's verify this with Why3 // International Journal on Software Tools for Technology Transfer. — 2015. — Vol. 17, Issue 6. — P. 709–727.
57. Böhme S., Moskal M., Schulte W., Wolff B. HOL-Boogie—An Interactive Prover-Backend for the Verifying C Compiler // Journal of Automated Reasoning. — 2010. — Vol. 44, Issue 1–2. — Article ID: 111.
58. Chakraborty S., Gupta A., Unadkat D. Diffy: Inductive Reasoning of Array Programs Using Difference Invariants // Lect. Notes Comput. Sci. — Cham, 2021. — Vol. 12760. — P. 911–935.
59. Cohen E., Dahlweid M., Hillebrand M. A., Leinenbach D., Moskal M., Santen T., Schulte W., Tobies S. VCC: A Practical System for Verifying Concurrent C // Lect. Notes Comput. Sci. — Berlin etc., 2009. — Vol. 5674. — P. 23—42.
60. Cohen E., Moskal M., Schulte W., Tobies S. Local Verification of Global Invariants in Concurrent Programs // Lect. Notes Comput. Sci. — Berlin etc., 2010. — Vol. 6174. — P. 480–494.
61. Cok D. R. Java Automated Deductive Verification in Practice: Lessons from Industrial Proof-Based Projects // Lect. Notes Comput. Sci. — Cham, 2018. — Vol. 11247. — P. 176–193.
62. Cok D. R. JML and OpenJML for Java 16 // Proc. 23rd ACM Intern. Workshop on Formal Techniques for Java-like Programs, Denmark, July 13, 2021. — New York: ACM, 2021. — P. 65–67.
63. Cok D. R. Reasoning about Functional Programming in Java and C++ // Companion Proc. ISSTA/ECOOP 2018 Workshops, Amsterdam, Netherlands,

July 16–21, 2018. — New York: ACM, 2018. — P. 37–39.

64. Cok D. R., Tasiran S. Practical Methods for Reasoning About Java 8's Functional Programming Features // Lect. Notes Comput. Sci. — Cham, 2018. — Vol. 11294. — P. 267–278.
65. Correnson L. Qed. Computing What Remains to Be Proved // Lect. Notes Comput. Sci. — Cham, 2014. — Vol. 8430. — P. 215–229.
66. Cuoq P., Monate B., Pacalet A., Prevosto V. Functional dependencies of C functions via weakest pre-conditions // International Journal on Software Tools for Technology Transfer. — 2011. — Vol. 13, Issue 5. — P. 405–417.
67. Dailler S., Hauzar D., Marché C., Moy Y. Instrumenting a weakest precondition calculus for counterexample generation // Journal of Logical and Algebraic Methods in Programming. — 2018. — Vol. 99. — P. 97–113.
68. Dailler S., Marché C., Moy Y. Lightweight interactive proving inside an automatic program verifier // Proc. Fourth Workshop on Formal Integrated Development Environment, Oxford, UK, July 14, 2018. — Electronic Proc. Theor. Comput. Sci. — 2018. — Vol. 284. — P. 1–15.
69. Davis J., Myreen M. O. The Reflective Milawa Theorem Prover is Sound (Down to the Machine Code that Runs it) // Journal of Automated Reasoning. — 2015. — Vol. 55, Issue 2. — P. 117–183.
70. de Angelis E., Fioravanti F., Pettorossi A., Proietti M. Proving Properties of Sorting Programs: A Case Study in Horn Clause Verification // Proc. HCVS/PERR 2019, Prague, Czech Republic, April 6-7, 2019. — Electronic Proc. Theor. Comput. Sci. — 2019. — Vol. 296. — P. 48–75.
71. Dean J., Ghemawat S. MapReduce: simplified data processing on large clusters // Communications of the ACM. — 2008. — Vol. 51, Issue 1. — P. 107—113.
72. de Gouw S., de Boer F. S., Bubel R., Hähnle R., Rot J., Steinhöfel D. Verifying OpenJDK's Sort Method for Generic Collections // Journal of Automated Reasoning. — 2019. — Vol. 62, Issue 1. — P. 93–126.
73. de Moura L., Bjørner N. Z3: An Efficient SMT Solver // Lect. Notes Comput. Sci. — Berlin etc., 2008. — Vol. 4963. — P. 337–340.
74. Denney E., Fischer B. Explaining Verification Conditions // Lect. Notes

Comput. Sci. — Berlin etc., 2008. — Vol. 5140. — P. 145–159.

75. Detlefs D., Nelson G., Saxe J.B. Simplify: a theorem prover for program checking // Journal of the ACM. — 2005. — Vol. 52, Issue 3. — P. 365–473.
76. Divasón J., Romero A. Using Krakatoa for Teaching Formal Verification of Java Programs // Lect. Notes Comput. Sci. — Cham, 2019. — Vol 11758. — P. 37–51.
77. Dongarra J. J., van der Steen A. J. High-performance computing systems: Status and outlook // Acta Numerica. — 2012. — Vol. 21. — P. 379–474.
78. Efremov D., Mandrykin M., Khoroshilov A. Deductive verification of unmodified Linux kernel library functions // Lect. Notes Comput. Sci. — Cham, 2018. — Vol. 11245. — P. 216–234.
79. Ernst G. Loop Verification with Invariants and Contracts // Lect. Notes Comput. Sci. — Cham, 2022. — Vol. 13182. — P. 69–92.
80. Ernst G., Huisman M., Mostowski W., Ulbrich M. VerifyThis — Verification Competition with a Human Factor // Lect. Notes Comput. Sci. — Cham, 2019. — Vol. 11429. — P. 176–195.
81. Feo J. T., Cann D. C., Oldehoeft R. R. A report on the sisal language project // Journal of Parallel and Distributed Computing. — 1990. — Vol. 10, Issue 4. — P. 349–366.
82. Filliâtre J. C. Deductive software verification // International Journal on Software Tools for Technology Transfer. — 2011. — Vol. 13, Issue 5. — Article ID: 397.
83. Filliâtre J.C., Marché C. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification // Lect. Notes Comput. Sci. — Berlin etc., 2007. — Vol. 4590. — P. 173—177.
84. Filliâtre J. C., Paskevich A. Why3 — Where Programs Meet Provers // Lect. Notes Comput. Sci. — Berlin etc., 2013. — Vol. 7792. — P. 125—128.
85. Floyd R. W. Assigning meanings to programs // Proc. Symposia in Applied Mathematics. — Providence, 1967. — Vol. 19. — P. 19–31.
86. Furia C.A., Meyer B., Velder S. Loop invariants: Analysis, classification, and examples // ACM Computing Surveys. — 2014. — Vol. 46, Issue 3. — Article

ID: 34. — P. 1–51.

87. Furia C.A., Nordio M., Polikarpova N., Tschannen J. AutoProof: auto-active functional verification of object-oriented programs // International Journal on Software Tools for Technology Transfer. — 2017. — Vol. 19, Issue 6. — P. 697–716.
88. Galeotti J. P., Furia C. A., May E., Fraser G., Zeller A. Inferring loop invariants by mutation, dynamic analysis, and static checking // IEEE Transactions on Software Engineering. — 2015. — Vol. 41, Issue 10. — P. 1019–1037.
89. Gargano M., Hillebrand M., Leinenbach D., Paul W. On the Correctness of Operating System Kernels // Lect. Notes Comput. Sci. — Berlin etc., 2005. — Vol. 3603. — P. 1–16.
90. Gaudiot J.-L., DeBoni T., Feo J., Böhm W., Najjar W., Miller P. The Sisal Project: Real World Functional Programming // Lect. Notes Comput. Sci. — Berlin etc., 2001. — Vol. 1808. — P. 45–72.
91. Grahl D., Ulbrich M. From Specification to Proof Obligations // Lect. Notes Comput. Sci. — Cham, 2016. — Vol. 10001. — P. 243–287.
92. Grebing S., Klamroth J., Ulbrich M. Seamless Interactive Program Verification // Lect. Notes Comput. Sci. — Cham, 2020. — Vol. 12031. — P. 68–86.
93. Hähnle R., Huisman M. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools // Lect. Notes Comput. Sci. — Cham, 2019. — Vol. 10000. — P. 345–373.
94. Heras J., Komendantskaya E., Johansson M., Maclean E. Proof-pattern recognition and lemma discovery in ACL2 // Lect. Notes Comput. Sci. — Berlin etc., 2013. — Vol. 8312. — P. 389–406.
95. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. — 1969. — Vol. 12, Issue. 10. — P. 576–580.
96. Hoare C. A. R., Misra J., Leavens G. T., Shankar N. The verified software initiative: A manifesto // ACM Computing Surveys. — 2009. — Vol. 41, Issue 4. — Article ID: 22. — P. 1–8.
97. Humenberger A., Jaroschek M., Kovács L. Invariant Generation for Multi-Path Loops with Polynomial Assignments // Lect. Notes Comput. Sci. — Cham,

2018. — Vol. 10747. — P. 226–246.

98. Hunt W. A., Kaufmann M., Moore J. S., Slobodova A. Industrial hardware and software verification with ACL2 // Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences. — 2017. — Vol. 375, Issue 2104. — Article ID: 20150399.
99. Imine A., Ranise S. Building Satisfiability Procedures for Verification: The Case Study of Sorting Algorithms // Preproc. LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003. — Leuven: KU Leuven, 2003. — P. 65–80.
100. Jacobs B., Kiniry J., Warnier M. Java Program Verification Challenges // Lect. Notes Comput. Sci. — Berlin etc., 2003. — Vol. 2852. — P. 202–219.
101. Jiang D., Zhou M. A comparative study of insertion sorting algorithm verification // Proc. 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference, Chengdu, China, December 15-17, 2017 — IEEE, 2017. — P. 321–325.
102. Johansson M. Lemma Discovery for Induction // Lect. Notes Comput. Sci. — Cham, 2019. — Vol. 11617. — P. 125–139.
103. Kasyanov V. Sisal 3.2: functional language for scientific parallel programming // Enterprise Information Systems. — 2013. — Vol. 7, Issue 2. — P. 227–236.
104. Kasyanov V., Kasyanova E. A System of Functional Programming for Supporting of Cloud Supercomputing // WSEAS Transactions on Information Science and Applications. — 2018. — Vol. 15. — P. 81–90.
105. Kasyanov V., Kasyanova E. Methods and System for Cloud Parallel Programming // Proceedings of the 21st International Conference on Enterprise Information Systems, Heraklion, Greece, May 3–5, 2019. — Setubal: SciTePress. — 2019. — P. 623–629.
106. Kasyanov V. N., Kasyanova E. V., Kondratyev D. A. Formal verification of Cloud Sisal programs // Proc. Applied Physics, Simulation and Computing, Rome, Italy, May 23–25, 2020. — Journal of Physics: Conference Series. — 2020. — Vol. 1603. — Article ID: 012020.
107. Kasyanov V. N., Kasyanova E. V., Malishev A. A. Support tools for functional programming distance learning and teaching // Proc. Intern. Conf. «Marchuk

Scientific Readings 2021», Novosibirsk, Russia, October 4–8, 2021. — Journal of Physics: Conference Series. — 2021. — Vol. 2099. — Article ID: 012052.

108. Kasyanov V. N., Stasenko A. P. Sisal 3.2 Language Structure Decomposition // Lecture Notes in Electrical Engineering. — Boston, 2009. — Vol. 28. — P. 533–543.

109. Kaufmann M., Moore J. S. A Precise Description of the ACL2 Logic. — Austin, 1998. — 74 p. — URL: https://www.cs.utexas.edu/users/moore/publications/km97a.pdf (дата обращения: 19.01.2022)

110. Kaufmann M., Moore J. S. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp // IEEE Transactions on Software Engineering. —1997. — Vol. 23, Issue 4. — P. 203–213.

111. Kohlhase M., Müller D., Owre S., Rabe F. Making PVS Accessible to Generic Services by Interpretation in a Universal Format // Lect. Notes Comput. Sci. — Cham, 2017. — Vol. 10499. — P. 319–335.

112. Kondratyev D. A. Automated Error Localization in C Programs [Электронный ресурс] // Режим доступа: https://bitbucket.org/Kondratyev/verify-c-light, свободный (дата обращения: 10.03.2022).

113. Kondratyev D. A. Automatic verification of insertion sorting [Электронный ресурс] // Режим доступа: https://bitbucket.org/Kondratyev/verify-loops, свободный (дата обращения: 11.03.2022).

114. Kondratyev D. Implementing the Symbolic Method of Verification in the C-Light Project // Lect. Notes Comput. Sci. — Cham, 2018. — Vol. 10742. — P. 227–240.

115. Kondratyev D. A. Towards loop invariant elimination for definite iterations over changeable data structures in C programs verification. Appendices [Электронный ресурс] // Режим доступа: https://bitbucket.org/c-light/loop-invariant-elimination, свободный (дата обращения: 10.03.2022).

116. Kondratyev D. A. Verification of Insertion Sorting Program [Электронный ресурс] // Режим доступа: https://bitbucket.org/Kondratyev/sorting, свобод-

ный (дата обращения: 15.03.2022).

117. Kondratyev D. A., Maryasov I. V., Nepomniaschy V. A. The Automation of C Program Verification by the Symbolic Method of Loop Invariant Elimination // Automatic Control and Computer Sciences. — 2019. — Vol. 53, Issue 7. — P. 653–662.
118. Kondratyev D., Maryasov I., Nepomniaschy V. Towards Automatic Deductive Verification of C Programs over Linear Arrays // Lect. Notes Comput. Sci. — Cham, 2019. — Vol. 11964. — P. 232–242.
119. Kondratyev D., Maryasov I., Nepomniaschy V. Towards Automatic Deductive Verification of C Programs Over Linear Arrays // Preliminary Proc. Intern. Conf. PSI–2019: A.P. Ershov Informatics Conf. "Perspectives of System Informatics", Novosibirsk, Russia, July 2–5, 2019. — Novosibirsk: Publishing center of Novosibirsk State University, 2019. — P. 162–171.
120. Kondratyev D. A., Maryasov I. V., Nepomniaschy V. A. Towards Loop Invariant Elimination for Definite Iterations over Changeable Data Structures in C Programs Verification. // Proc. 9th Workshop "Program Semantics, Specification and Verification: Theory and Applications", Yaroslavl, Russia, June 21–22, 2018. — Yaroslavl: Yaroslavl Demidov State University, 2018. — P. 51–57.
121. Kondratyev D. A., Nepomniaschy V. A. Automation of C-program deductive verification without using loop invariants // Programming and Computer Software. — 2022. — Vol. 47, Issue 5 (To appear).
122. Kondratyev D., Promsky A. Automated Sisal program verification with ACL2 // Preliminary Proc. Intern. Conf. PSI–2019: A.P. Ershov Informatics Conf. "Perspectives of System Informatics", Novosibirsk, Russia, July 2–5, 2019. — Novosibirsk: IPC NSU, 2019. — P. 172–185.
123. Kondratyev D., Promsky A. Correctness of Proof Strategy for the Sisal Program Verification // Proc. 2019 Intern. Multi-Conf. Eng., Comput. and Inf. Sci., Novosibirsk, Russia, October 21-27, 2019. — IEEE, 2019. — P. 641–646.
124. Kondratyev D. A., Promsky A. V. Developing a Self Applicable Verification System. Theory and Practices // Automatic Control and Computer Sciences.

— 2015. — Vol. 49, Issue 7. – P. 445–452.

125. Kondratyev D., Promsky A. Proof Strategy for Automated Sisal Program Verification // Lect. Notes Comput. Sci. — Cham, 2019. — Vol. 11771. — P. 113–120.
126. Kondratyev D. A., Promsky A. V. The Complex Approach of the C-lightVer System to the Automated Error Localization in C-Programs // Automatic Control and Computer Sciences. — 2020. — Vol. 54, Issue 7. — P. 728–739.
127. Kondratyev D. A., Promsky A. V. Towards automated error localization in C programs with loops // Abstracts of the 10th Workshop "Program Semantics, Specification and Verification: Theory and Applications". — Novosibirsk, Russia, July 1–2, 2019. — Novosibirsk: IPC NSU, 2019. — P. 24.
128. Kondratyev D. A., Promsky A. V. Towards automated error localization in C programs with loops // System Informatics. — 2019. —Issue 14. — P. 31–44.
129. Kondratyev D. A., Promsky A. V. Towards the 'verified verifier'. Theory and practice // Proc. Fifth Workshop "Program Semantics, Specification and Verification: Theory and Applications", Moscow, Russia, June 6, 2014. — P. 68–78.
130. Kondratyev D. A., Promsky A. V. Towards verification of scientific and engineering programs. The CPPS project // Journal of Computational technologies. — 2020. — Vol. 25, Issue 5. — P. 91–106.
131. Könighofer R., Toegl R., Bloem R. Automatic Error Localization for Software Using Deductive Verification // Lect. Notes Comput. Sci. — Cham, 2014. — Vol. 8855. — P. 92–98.
132. Kosmatov N., Marché C., Moy Y., Signoles J. Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014 // Lect. Notes Comput. Sci. — Cham, 2016. — Vol. 9952. — P. 461–478.
133. Kostyukov Y., Mordvinov D., Fedyukovich G. Beyond the elementary representations of program invariants over algebraic data types // Proc. 42nd ACM SIGPLAN Intern. Conf. on Programming Language Design and Implementation, Canada, June 20-25, 2021. — New York: ACM, 2021. — P. 451–465.

134. Kovács L. Symbolic Computation and Automated Reasoning for Program Analysis // Lect. Notes Comput. Sci. — Cham, 2016. — Vol. 9681. — P. 20–27.
135. Krebbers R., Wiedijk F. A Typed C11 Semantics for Interactive Theorem Proving // Proc. 2015 Conf. Certified Programs and Proofs, Mumbai, India, January 13-14, 2015. — New York: ACM, 2015. — P. 15–27.
136. Leinenbach D., Paul W., Petrova E. Towards the formal verification of a C0 compiler: code generation and implementation correctness // Proc. Third IEEE Intern. Conf. on Software Engineering and Formal Methods., Koblenz, Germany, September 7-9, 2005. — IEEE, 2005. — P. 2–11.
137. Leino K. R. M. Accessible Software Verification with Dafny // IEEE Software. — 2017. — Vol. 34. — Issue. 6. — P. 94–97.
138. Leino K. R. M. Automating Induction with an SMT Solver // Lect. Notes Comput. Sci. — Berlin etc., 2012. — Vol. 7148. — P. 315–331.
139. Leino K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness // Lect. Notes Comput. Sci. — Berlin etc., 2010. — Vol. 6355. — P. 348—370.
140. Leino K. R. M. Program proving using intermediate verification languages (IVLs) like Boogie and Why3 // Proc. 2012 ACM conference on High integrity language technology, Boston, USA, December 2-6, 2012.— New York: ACM, 2012. — P. 25–26.
141. Leino K. R. M., Rümmer P. A Polymorphic Intermediate Verification Language: Design and Logical Encoding // Lect. Notes Comput. Sci. — Berlin etc., 2010. — Vol. 6015. — P. 312—327.
142. Leroy X. Formal verification of a realistic compiler // Communications of the ACM. — 2009. — Vol. 52, Issue 7. — P. 107–115.
143. Limperg J. A novice-friendly induction tactic for lean // Proc. 10th ACM SIGPLAN Intern. Conf. on Certified Programs and Proofs, Denmark, January 17-19, 2021. — New York: ACM, 2021. — P. 199–211.
144. Maryasov I. V., Nepomniaschy V. A., Kondratyev D. A. Invariant Elimination of Definite Iterations over Arrays in C Programs Verification // Modeling and Analysis of Information Systems. — 2017. — Vol. 24, Issue 6. — P. 743–754.

145. Maryasov I. V., Nepomniaschy V. A., Kondratyev D. A. Verification of Definite Iteration over Arrays with a Loop Exit in C Programs // Abstracts of the 8th Workshop "Program Semantics, Specification and Verification: Theory and Applications". — Moscow, Russia, June 26, 2017. — Moscow: MAKS Press, 2017. — P. 10.
146. Maryasov I. V., Nepomniaschy V. A., Kondratyev D. A. Verification of Definite Iteration over Arrays with a Loop Exit in C Programs. // System Informatics. — 2017. — Issue 10. — P. 57–66.
147. Maryasov I. V., Nepomniaschy V. A., Promsky A. V., Kondratyev D.A. Automatic C Program Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. — 2014. — Vol. 48, Issue 7. — P. 407—414.
148. Maryasov I. V., Nepomniaschy V. A., Promsky A. V., Kondratyev D. A. Automatic C Program Verification Based on Mixed Axiomatic Semantics // Proc. Fourth Workshop "Program Semantics, Specification and Verification: Theory and Applications", Yekaterinburg, Russia, June 24, 2013. — Yaroslavl: Yaroslavl Demidov State University, 2013. — P. 50–59.
149. Möller B., O'Hearn P., Hoare T. On Algebra of Program Correctness and Incorrectness // Lect. Notes Comput. Sci. — Cham, 2021. — Vol. 13027. — P. 325–343.
150. Moore J. S. Milestones from the Pure Lisp theorem prover to ACL2 // Formal Aspects of Computing. — 2019. — Vol. 31, Issue 6. — P. 699–732.
151. Moriconi M., Schwarts R. L. Automatic Construction of Verification Condition Generators From Hoare Logics // Lect. Notes Comput. Sci. — Berlin etc., 1981. — Vol. 115. — P. 363–377.
152. Moskal M. From C to Infinity and Back: Unbounded Auto-active Verification with VCC // Lect. Notes Comput. Sci. — Berlin etc., 2012. — Vol. 7358. — P. 6–6.
153. Moskal M. Verifying Functional Correctness of C Programs with VCC // Lect. Notes Comput. Sci. — Berlin etc., 2011. — Vol. 6617. — P. 56–57.
154. Myreen M. O., Gordon M. J. C. Transforming programs into recursive functions

// Electronic Notes in Theoretical Computer Science. — 2009. — Vol. 240. — P. 185–200.

155. Nawaz M. S., Malik M., Li Y., Sun M., Lali M. I. U. A Survey on Theorem Provers in Formal Methods. — 2019. — 26 p. — URL: https://arxiv.org/abs/1912.03028 (дата обращения: 21.03.2022)
156. Nepomniaschy V. A., Anureev I. S., Mikhailov I. N., Promskii A. V. Towards verification of C programs. C-light language and its formal semantics // Programming and Computer Software. — 2002. — Vol. 28, Issue 6. — P. 314–323.
157. Nepomniaschy V. A., Anureev I. S., Promskii A. V. Towards Verification of C Programs: Axiomatic Semantics of the C-kernel Language // Programming and Computer Software. — 2003. — Vol. 29, Issue 6. — P. 338–350.
158. Nepomniaschy V. A. Symbolic method of verification of definite iterations over altered data structures // Programming and Computer Software. — 2005. — Vol. 31, Issue 1. — P. 1–9.
159. Nordio M., Calcagno C., Müller P., Meyer B. A Sound and Complete Program Logic for Eiffel // Lecture Notes in Business Information Processing. — Berlin etc., 2009. — Vol. 33. — P. 195–214.
160. Owre S., Rushby J. M., Shankar N. PVS: A prototype verification system // Lect. Notes Comput. Sci. — Berlin etc., 1992. — Vol. 607. — P. 748–752.
161. Paulin-Mohring C. Introduction to the Coq Proof-Assistant for Practical Software Verification // Lect. Notes Comput. Sci. — Berlin etc., 2012. — Vol. 7682. — P. 45–95.
162. Paulson L. C., Nipkow T., Wenzel M. From LCF to Isabelle/HOL // Formal Aspects of Computing. — 2019. — Vol. 31, Issue 6. — P. 675–698.
163. Polikarpova N., Tschannen J., Furia C. A. A fully verified container library // Formal Aspects of Computing. — 2018. — Vol. 30, Issue 5. — P. 495–523.
164. Programming languages — C. — ISO/IEC 9899:1999. — 538 p.
165. Programming languages — C. — ISO/IEC 9899:2011. — 683 p.
166. Programming languages — C++. — ISO/IEC 14882:2020. — 1853 p.
167. Promsky A. V., Kondratyev D. A. Implementing the MetaVCG approach in

the C-light system // Proc. 3rd Internat. workshop-conf. Tools & Methods of Program Analysis, St. Petersburg, Russia, November 12–14, 2015. — St. Petersburg: Peter the Great St. Petersburg Polytechnic University, 2015. — P. 101–106.

168. Pyzhov K., Idrisov R. Back-end translator for Sisal 3.1 compiler // Bulletin of the Novosibirsk Computing Center. Series: Comput. Sci. — 2013. — Issue 35. — P. 101–119.
169. Raad A., Berdine J., Dang H.H., Dreyer D., O'Hearn P., Villard J. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic // Lect. Notes Comput. Sci. — Cham, 2020. — Vol. 12225. — P. 225–252.
170. Reger G., Voronkov A. Induction in Saturation-Based Proof Search // Lect. Notes Comput. Sci. — Cham, 2019. — Vol. 11716. — P. 477–494.
171. Reynolds A., Kuncak V. Induction for SMT Solvers // Lect. Notes Comput. Sci. — Berlin etc., 2015. — Vol. 8931. — P. 80–98.
172. Safari M., Huisman M. A Generic Approach to the Verification of the Permutation Property of Sequential and Parallel Swap-Based Sorting Algorithms // Lect. Notes Comput. Sci. — Cham, 2020. — Vol 12546. — P. 257–275.
173. Sammler M., Lepigre R., Krebbers R., Memarian K., Dreyer D., Garg D. RefinedC: automating the foundational verification of C code with refined ownership types // Proc. 42nd ACM SIGPLAN Internat. Conf. on Programming Language Design and Implementation, June 20-25, 2021. — New York: ACM, 2021. — P. 158–174.
174. Schmitt P. H. A Short History of KeY // Lect. Notes Comput. Sci. — Cham, 2020. — Vol. 12345. — P. 3–18.
175. Srivastava S., Gulwani S., Foster J. S. Template-based program verification and program synthesis // International Journal on Software Tools for Technology Transfer. — 2013. — Vol. 15, Issue 5–6. — P. 497–518.
176. Stasenko A. Sisal 3.2 Language Features Overview // Lect. Notes Comput. Sci. — Berlin etc., 2011. — Vol. 6873. — P. 110–124.
177. Tschannen J., Furia C. A., Nordio M., Meyer B. Automatic Verification of

Advanced Object-Oriented Features: The AutoProof Approach // Lect. Notes Comput. Sci. — Berlin etc., 2012. — Vol. 7682. — P. 133—155.

178. Tuerk T. Local reasoning about while-loops // Proc. Theory Workshop at VSTTE 2010. Edinburgh, UK, August 18, 2010. — Zürich: ETH Zürich, 2010. — P. 29–39.

179. Ulbrich M. Dynamic Logic for an Intermediate Language: Verification, Interaction and Refinement. Berlin: epubli GmbH, 2014 — P. 268.

180. Vediramana Krishnan H. G., Shoham S., Gurfinkel A. Solving constrained Horn clauses modulo algebraic data types and recursive functions // Proceedings of the ACM on Programming Languages. — 2022. — Vol. 6, Issue POPL. — Article ID: 60. — P. 1–29.

181. Volkov G., Mandrykin M., Efremov D. Lemma Functions for Frama-C: C Programs as Proofs // Proc. 2018 Ivannikov Ispras Open Conf., Moscow, Russia, November 22-23, 2018. — IEEE, 2018. — P. 31–38.

Список сокращений и условных обозначений

- УК — условия корректности;
- VC — verification conditions, условия корректности;
- C-lightVer — C-light Verification, система верификации C-light программ;
- ГУК — генератор условий корректности;
- VCG — verification condition generator, генератор условий корректности;
- МГУК — метагенератор условий корректности;
- MetaVCG — verification condition metagenerator, метагенератор условий корректности;
- WP — weakest precondition, слабейшее предусловие;
- ACSL — ANSI/ISO C Specification Language, язык спецификации программ на языке ANSI/ISO C;
- SMT — satisfiability modulo theories, задача определения выполнимости формулы языка первого порядка в некоторой теории;
- ACL — Applicative Common Lisp, аппликативный диалект языка Common Lisp;
- ACL2 — A Computational Logic for Applicative Common Lisp, система вычислимой логики для аппликативного диалекта языка Common Lisp;
- CPPS — Cloud Parallel Programming System, система облачного параллельного программирования;
- CSV1 — Cloud Sisal Verification 1, первый модуль верификации Cloud Sisal программ;
- CSV2 — Cloud Sisal Verification 2, второй модуль верификации Cloud Sisal программ.