MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 4, 2021
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Towards Automatic Deductive Verification of C Programs with Sisal Loops Using the C-lightVer System

D. A. Kondratyev[1]

DOI: 10.18255/1818-1015-2021-4-372-393

[1]A.P. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences, 6, Acad. Lavrentjev pr., Novosibirsk 630090, Russia.

MSC2020: 68Q60
Research article
Full text in Russian

Received November 15, 2021
After revision December 1, 2021
Accepted December 8, 2021

The C-lightVer system is developed in IIS SB RAS for C-program deductive verification. C-kernel is an intermediate verification language in this system. Cloud parallel programming system (CPPS) is also developed in IIS SB RAS. Cloud Sisal is an input language of CPPS. The main feature of CPPS is implicit parallel execution based on automatic parallelization of Cloud Sisal loops. Cloud-Sisal-kernel is an intermediate verification language in the CPPS system. Our goal is automatic parallelization of such a superset of C that allows implementing automatic verification. Our solution is such a superset of C-kernel as C-Sisal-kernel. The first result presented in this paper is an extension of C-kernel by Cloud-Sisal-kernel loops. We have obtained the C-Sisal-kernel language. The second result is an extension of C-kernel axiomatic semantics by inference rule for Cloud-Sisal-kernel loops. The paper also presents our approach to the problem of deductive verification automation in the case of finite iterations over data structures. This kind of loops is referred to as definite iterations. Our solution is a composition of symbolic method of verification of definite iterations, verification condition metageneration and mixed axiomatic semantics method. Symbolic method of verification of definite iterations allows defining inference rules for these loops without invariants. Symbolic replacement of definite iterations by recursive functions is the base of this method. Obtained verification conditions with applications of recursive functions correspond to logical base of ACL2 prover. We use ACL2 system based on computable recursive functions. Verification condition metageneration allows simplifying implementation of new inference rules in a verification system. The use of mixed axiomatic semantics results to simpler verification conditions in some cases.

Keywords: deductive verification; C-lightVer; cloud parallel programming system; symbolic method of verification of definite iterations; loop invariant; ACL2

INFORMATION ABOUT THE AUTHORS

Dmitry A. Kondratyev | orcid.org/0000-0002-9387-6735. E-mail: apple-66@mail.ru
correspondence author | Junior researcher.

Funding: RSF, project No 18-11-00118.

For citation: D. A. Kondratyev, "Towards Automatic Deductive Verification of C Programs with Sisal Loops Using the C-lightVer System", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 372-393, 2021.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 4, 2021
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

На пути к автоматической дедуктивной верификации С-программ с Sisal-циклами в системе C-lightVer

Д. А. Кондратьев[1]

DOI: 10.18255/1818-1015-2021-4-372-393

[1]Институт систем информатики им. А.П. Ершова Сибирского отделения Российской академии наук, проспект Академика Лаврентьева, д. 6, г. Новосибирск, 630090 Россия.

УДК 004.052.42
Научная статья
Полный текст на русском языке

Получена 15 ноября 2021 г.
После доработки 1 декабря 2021 г.
Принята к публикации 8 декабря 2021 г.

В Институте систем информатики СО РАН разрабатывается система C-lightVer для дедуктивной верификации С-программ. C-kernel является промежуточным языком верификации в данной системе. Система облачного параллельного программирования (CPPS) также разрабатывается в Институте систем информатики СО РАН. Cloud Sisal является входным языком системы CPPS. Главной особенностью системы CPPS является неявное параллельное исполнение, основанное на автоматическом распараллеливании циклов Cloud Sisal. Cloud-Sisal-kernel является промежуточным языком верификации в системе CPPS. Нашей целью является автоматическое распараллеливание такого надмножества языка С, которое позволяет реализовать автоматическую верификацию. Нашим решением является такое надмножество языка C-kernel, как язык C-Sisal-kernel. Первым результатом, представленным в данной статье, является расширение языка C-kernel циклами языка Cloud-Sisal-kernel. В результате был разработан язык C-Sisal-kernel. Вторым результатом, представленным в данной статье, является расширение аксиоматической семантики языка C-kernel правилом вывода для циклов языка Cloud-Sisal-kernel. В данной статье также представлен наш подход к проблеме автоматизации дедуктивной верификации в случае финитных итераций над структурами данных. Такие циклы называются финитными итерациями. Нашим решением является композиция символического метода верификации финитных итераций, метагенерации условий корректности и смешанной аксиоматической семантики. Символический метод верификации финитных итераций позволяет задавать правила вывода для таких циклов без инвариантов. Символическая замена финитных итераций рекурсивными функциями является основой данного метода. Полученные условия корректности с применениями рекурсивных функций соответствуют логической основе системы доказательства ACL2. Мы используем систему ACL2, основанную на вычислимых рекурсивных функциях. Метагенерация условий корректности позволяет упростить реализацию новых правил вывода в системе верификации. Использование смешанной аксиоматической семантики приводит в некоторых случаях к более простым условиям корректности.

Ключевые слова: дедуктивная верификация; C-lightVer; система облачного параллельного программирования; символический метод верификации финитных итераций; инвариант цикла; ACL2

ИНФОРМАЦИЯ ОБ АВТОРАХ

Дмитрий Александрович Кондратьев
автор для корреспонденции

orcid.org/0000-0002-9387-6735. E-mail: apple-66@mail.ru
Младший научный сотрудник.

Финансирование: РНФ, проект № 18-11-00118.

Для цитирования: D. A. Kondratyev, “Towards Automatic Deductive Verification of C Programs with Sisal Loops Using the C-lightVer System”, *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 372-393, 2021.

Введение

Система C-lightVer [1—5] разрабатывается в Институте систем информатики СО РАН для верификации С-программ. Данная система основана на методе дедуктивной верификации [6—9]. Представительное подмножество языка С (C-light) [10] является входным языком системы C-lightVer. Модель памяти языка C-light основана на отображениях MeM и MD. MeM является отображением из имен объектов в их адреса, MD является отображением из адресов объектов в их значения. Операция upd позволяет создавать новое отображение MD, когда изменяется состояние памяти. C-kernel [11] является промежуточным языком верификации в системе C-lightVer. Трансляция из C-light в C-kernel основана на наборе правил [1]. Главной целью данной трансляции является локализация побочных эффектов.

Символический метод верификации финитных итераций [12] применяется к циклам специального вида (финитным итерациям). Тело финитной итерации исполняется один раз для каждого элемента структуры данных конечной размерности. Основой этого метода является символическая замена финитных итераций специальными рекурсивными функциями. Данный метод позволяет избежать задания инвариантов в случае финитных итераций. Система C-lightVer применяет этот метод к циклам языка С, соответствующим финитным итерациям [3—5].

Метагенерация условий корректности (УК) [2, 5, 13] позволяет использовать правила вывода УК как входные данные системы верификации. Входными данными метагенератора являются правила вывода УК и аннотированная программа. Заключения правил вывода представляют шаблоны, сопоставляемые с кодом на языке С. Поэтому, был разработан язык шаблонов [2] для задания правил вывода.

Метод смешанной аксиоматической семантики [1] позволяет использовать специальные версии правил вывода для определенных программных конструкций.

В системе C-lightVer для доказательства УК используется ACL2 [14]. Applicative Common Lisp (ACL) является входным языком системы ACL2. Система C-lightVer генерирует УК, записанные на языке ACL. Система ACL2 основана на вычислимых рекурсивных функциях [14]. Полученные УК с применениями рекурсивных функций соответствуют логической основе системы ACL2.

Важным этапом автоматизации верификации является этап автоматизации доказательства УК. Доказательство УК, содержащих операцию замены для финитной итерации, основано на индукции по длине структуры, над которой осуществляется данная итерация. При этом, использования классической индукции в системе ACL2 не достаточно для доказательства УК, содержащих операцию замены для финитной итерации, в случае, если итерация изменяет элементы структуры или содержит инструкции, подобные `break`. В системе C-lightVer была реализована стратегия автоматизации доказательства УК программ, постусловием которых является разбор случаев [4].

Система облачного параллельного программирования (CPPS) также разрабатывается в Институте систем информатики СО РАН [15]. Cloud Sisal [15—18] является входным языком системы CPPS. Данный язык является новой версией языка Sisal [19, 20]. 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 [15, 16, 21]. Cloud-Sisal-kernel [22] был разработан как промежуточный язык верификации для дедуктивной верификации Cloud Sisal программ. Автоматизация верификации циклов Cloud-Sisal-kernel является результатом использования символического метода верификации финитных итераций [12].

Нашей целью является автоматическое распараллеливание такого надмножества С, которое позволяет реализовать автоматическую верификацию. Нашим решением является такое надмножество C-kernel, как C-Sisal-kernel. Это расширение C-kernel циклами языка Cloud-Sisal-kernel. Реализация неявного параллельного исполнения программ на языке C-Sisal-kernel основано на опыте [15, 16, 21]

распараллеливания программ на языке Cloud Sisal. Отметим, что это задача другой группы исследователей [15—17, 21] из Института систем информатики СО РАН. Эта группа разрабатывает CPPS. Автоматическое распараллеливание C-Sisal-kernel программ является частью проекта CPPS.

Нашей задачей является расширение аксиоматической семантики языка C-kernel для реализации дедуктивной верификации C-программ с циклами языка Sisal. Дедуктивная верификация C-Sisal-kernel программ является частью системы C-lightVer. Мы применяем такие подходы как метод метагенерации УК [2, 5] и метод смешанной аксиоматической семантики [1] для решения этой задачи.

Первым результатом, представленным в данной статье, является расширения языка C-kernel циклами Cloud-Sisal-kernel. Полученный язык называется C-Sisal-kernel. Вторым результатом, представленным в данной статье, является расширение аксиоматической семантики C-kernel правилом вывода для циклов языка Cloud-Sisal-kernel. Эта семантика позволяет автоматизировать дедуктивную верификацию благодаря отсутствию инвариантов циклов. Данная статья также представляет наш комплексный подход к проблеме автоматизации дедуктивной верификации в случае финитных итераций над структурами данных. Нашим решением является композиция символического метода верификации финитных итераций [4, 5, 12], метода метагенерации УК [2, 5, 13] и метода смешанной аксиоматической семантики [1]. Мы полагаем, что наш подход может быть применен к различным языкам программирования, которые позволяют записывать программы с циклами, соответствующими финитным итерациям.

Данная статья имеет следующую структуру: предварительные сведения описаны в главах 1, 2 и 3, новые результаты описаны в главе 4 и эксперименты, демонстрирующие применение новых результатов, описаны в главе 5.

Обзор родственных работ. Во-первых, специальные виды циклов являются важной концепцией в различных парадигмах программирования. Например, функции *map* и *reduce* реализованы во многих функциональных языках программирования. Библиотека MapReduce [23] расширяет императивные и объектно-ориентированные языки программирования конструкциями, соответствующими функциональной парадигме программирования. Другим примером является специальная конструкция *loop* в языке Applicative Common Lisp [24]. Операционная семантика для расширения языка C-like конструкциями OpenMP была предложена в работе [25]. 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 [26]. Мы полагаем, что символический метод верификации финитных итераций может быть применен к различным итерациям, упомянутым в этом разделе, чтобы избежать задания их инвариантов.

Во-вторых, расширение языков программирования конструкциями функциональной парадигмы программирования является актуальной задачей. Такие конструкции могут увеличить эффективность исполнения и упростить реализацию некоторых алгоритмов. Например, семантика конструкций функциональной парадигмы программирования в новейших стандартах C++ и Java описана в работах [27, 28].

В третьих, расширения языка C специальными видами циклов является актуальной задачей. Новейший стандарт C++20 [29] вводит диапазоны (ranges) и итерации над ними. Более того, диапазоны (ranges) напоминают триплеты языка Cloud Sisal. Но циклы *for* над этими диапазонами (ranges) не позволяют, в отличие от Sisal-циклов, использовать редукции. Отметим, что современный стандарт C11 [30] был расширен новыми конструкциями с формальной семантикой [31], но эти конструкции не являются видами итераций. Система RefinedC [32] использует эту семантику, реализованную в системе доказательства Coq, но пользователю RefinedC необходимо задавать инварианты циклов.

Наконец, верификация циклов, основанная на их замене рекурсивными функциями, представляет большой интерес для нас. Это актуальный подход, позволяющий избежать задания инвариантов циклов. Работы [33, 34] основаны на этом подходе. Другим примером данного подхода является символический метод верификации финитных итераций [12]. Этот метод был применен к специальным видам циклов for из языка C [3—5].

Главным альтернативным подходом является генерация инвариантов циклов. Например, автоматическая генерация инвариантов для P-разрешимых циклов была предложена в работе [35]. Правые части инструкций присваивания в теле P-разрешимого цикла должны иметь вид присваивания полиномиального выражения. Но некоторые Sisal-циклы не являются P-разрешимыми из-за неполиномиальных редукций и конструкций, подобных `break`. Другим примером является метод разностных инвариантов, реализованный в системе Diffy [36]. Но этот метод, в отличие от символического метода верификации финитных итераций, не подходит для циклов с инструкциями `break`. Другим подходом является задание предусловий и постусловий вместо инвариантов циклов [37]. В работе [38] также описано использование спецификаций специального вида вместо инвариантов циклов. Этот метод реализован в системе Frama-C [39]. Но эти методы [37, 38] основаны на задании спецификаций пользователем. Также есть другие альтернативные походы. Одним из них является натуральная семантика. Такая семантика была предложена для Sisal-циклов в работе [40]. Но предложенная натуральная семантика для языка Sisal больше подходит для разработки компиляторов, чем для дедуктивной верификации. Другой альтернативой является использование трансляционной семантики. Например, работа [41] основана на трансляции циклов языка Cloud Sisal в циклы языка C. Трансформационная семантика была предложена для MapReduce [42]. Однако мы полагаем, что аксиоматическая семантика является лучшим выбором для дедуктивной верификации, поэтому эти альтернативы не используются в представленном исследовании.

1. Языки, методы и модули, используемые в системе C-lightVer

1.1. Язык C-light и язык C-kernel

Входным языком системы C-lightVer является язык C-light [10]. Этот язык является представительным подмножеством языка C. 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.

Операция upd позволяет создавать новое отображение MD, когда изменяется состояние памяти. Определим значение выражения $upd(MD,\ addr,\ val)$, где MD является отображением $address \longrightarrow value$, $addr$ 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 $(adr\ val)$.

Язык C-kernel является очень ограниченным подмножеством языка C-light [11]. 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.

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. УК полученной C-kernel программы генерируются на втором этапе исполнения системы C-lightVer.

1.2. Метагенератор условий корректности

Метагенератор УК является важным модулем системы C-lightVer. Входными данными метагенератора являются правила вывода УК и аннотированная программа. Заключения правил вывода представляют шаблоны, сопоставляемые с кодом на языке C. Поэтому, был разработан язык шаблонов [2] для задания правил вывода. Этот язык основан на логике первого порядка и грамматике языка C.

Правила вывода могут содержать нетерминальные символы, такие как неинтерпретированные предикатные символы или “фрагментные переменные”, обозначающие фрагменты кода [13]. Нетерминальный символ задает метаданные об определенной конструкции, соответствующей этому символу. Следовательно, в языке шаблонов есть специальные конструкции для представления этих метаданных. Например, конструкция `any_code(S)` может быть сопоставлена с любой последовательностью (включая пустую) инструкций на языке C. Специальный алгоритм [2] для сопоставления этих конструкций с аннотированным исходным кодом был реализован в метагенераторе. УК для аннотированной программы генерируются метагенератором с помощью использования входных правил вывода.

1.3. Символический метод верификации финитных итераций

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: **for x in S do v** := **body**(**v**, **x**), 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 $x \in S$. Пусть v_0 обозначает начальные значения переменных из v. Определим операцию замены $rep(v, S, body)$ для цикла:

1. Если $empty(S)$, тогда $rep(v_0, S, body) = v_0$;
2. Если $\neg empty(S)$, тогда $rep(v_0, S, body) = body(rep(v_0, rest(S), body), choo(S))$.

В случае наличия инструкции `break` в финитной итерации предложено следующее решение: когда происходит выход из цикла из-за исполнения этой инструкции, мы полагаем, что итерации цикла продолжаются, но значение v остается неизменным. Полученная операция замены возвращает не только вектор v, но также структуру с булевским полем (*loop-break*). Значением по умолчанию поля *loop-break* является $false$. Полученная функция *rep* содержит условие исполнения этой инструкции `break`. Если данная инструкция `break` исполняется, тогда эта функция *rep* возвращает структуру со значением $true$ поля *loop-break*. Также эта функция *rep* проверяет значение поля *loop-break* из результата рекурсивного вызова. Если значением является $true$, тогда эта функция *rep* возвращает тот же результат, что и рекурсивный вызов. Значения переменных цикла не изменяются после исполнения инструкции `break` в этой реализации.

Этот метод позволяет избежать задания инвариантов в случае циклов, соответствующих финитным итерациям [12]. Например, циклы языка Cloud-Sisal-kernel соответствуют финитным итерациям [22]. Другим примером является специальный класс циклов for в языке C. Эти циклы могут содержать инструкции `break`. Этот класс называется допустимыми циклами [4]. Алгоритм проверки, принадлежит ли определенный цикл этому классу, реализован в системе C-lightVer. Но, конечно, этот класс не покрывает все возможные финитные итерации в языке C. Генерация операции замены для допустимых циклов языка C-kernel основана на трансляторе $c2acl2$ [5] из C-kernel в ACL [14].

1.4. Метод смешанной аксиоматической семантики

Метод смешанной аксиоматической семантики [1] позволяет использовать специальные версии правил вывода для определенных программных конструкций. Отметим, что в C-программах есть переменные, которые используются без применений операций 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.

1.5. Стратегии автоматизации доказательства УК

Рассмотрим стратегию автоматизации доказательства УК программ, постусловием которых является разбор случаев [4]. Эта стратегия применяется для содержащих финитные итерации аннотированных программ, постусловие которых имеет вид конъюнкции импликаций. Для каждой такой импликации полезно рассмотреть, эквивалентен ли описанный ее посылкой случай исполнению операции `break`.

Рассматриваемая стратегия генерирует леммы, где в посылку УК добавляется специальный конъюнкт. Такой конъюнкт генерируется для каждой импликации постусловия в двух видах: для значения $rep(\ldots).loop\text{-}break$ и для отрицания значения $rep(\ldots).loop\text{-}break$. Данный конъюнкт является проверкой эквивалентности посылки импликации постусловия и значения/отрицания значения $rep(\ldots).loop\text{-}break$. Если система ACL2 автоматически доказывает какую-либо из сгенерированных лемм, то эта лемма добавляется в теорию предметной области. Система ACL2 может использовать леммы, добавленные в теорию предметной области, для доказательства других теорем и УК.

Так как значение поля $rep(\ldots).loop\text{-}break$ является информацией о срабатывании инструкции `break`, то данная стратегия разработана для такой аннотированной программы, где в постусловии описаны случаи срабатывания/отсутствия срабатывания инструкции `break`.

2. Краткое описание языка Cloud-Sisal-kernel

Язык Cloud Sisal основан на специальных видах циклов [15, 17, 18]. Эти виды циклов позволяют упростить распараллеливание и векторизацию операций над массивами [15, 16, 21]. Язык Cloud-Sisal-kernel был разработан как промежуточный язык верификации для дедуктивной верификации Cloud Sisal программ [22]. Для языка Cloud-Sisal-kernel была предложена аксиоматическая семантика. Этот язык включает базовые конструкции Cloud Sisal, такие как триплеты, циклические выражения, циклические выражения с условиями завершения и выражения замещения элементов массивов.

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.

2.1. Циклические выражения

Рассмотрим циклическое выражение, управляемое диапазоном:

$for\ var_1\ in\ triplet_1\ cross\ \ldots\ var_n\ in\ triplet_n\ do\ returns\ reduction\ expr\ end\ for$

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:

- $array\ (value)\ of$ возвращает массив (последнее значение) редуцируемого выражения;
- $sum\ (product)\ of$ вычисляет сумму (произведение) значений редуцируемых выражений;
- $greatest\ (least)\ of$ вычисляет наибольшее (наименьшее) значение редуцируемого выражения.

Обозначим как $loop\_e$ такое циклическое выражение с такими подвыражениями.

Рассмотрим иллюстративный пример Cloud Sisal программы:

```
function f (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 if (a[i, j] > 0) then a[i, j]*a[i, j]*a[i, j] else 0
                 end if end for end function
```

где

- `0..n-1..1` является конечной арифметической прогрессией, где `0` является начальным значением, `1` является шагом прогрессии и `n-1` является верхней границей;
- `0..m-1..1` является подобной последовательностью, где `m-1` является верхней границей;
- `sum of` является редукцией, которая вычисляет сумму редуцируемых выражений;
- `if (a[i,j] > 0) then a[i,j]*a[i,j]*a[i,j] else 0 end if` является редуцируемым выражением, которое вычисляет значение куба положительного элемента матрицы.

Эта функция вычисляет сумму значений кубов положительных элементов матрицы.

2.2. Циклические выражения с условиями завершения

Рассмотрим циклическое выражение с конструкцией $while$:

$for\ var_1\ in\ triplet_1\ cross\ \ldots\ var_n\ in\ triplet_n$
$while\ condition\ do\ returns\ reduction\ expr\ end\ for$

где конструкция $while$ соответствует следующей инструкции, записанной на языке программирования C: if (!$condition$) {**break**; }. Эта конструкция не задает вложенный цикл.

2.3. Выражения замещения элементов массивов

Выражение замещения элементов массива имеет следующий вид:

$$source\_array[var_1\ \texttt{in}\ triplet_1\ \texttt{cross}\ \ldots\ \texttt{cross}\ var_n\ \texttt{in}\ triplet_n\ :=\ expr]$$

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 замещение 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.

3. Семантика языка Cloud-Sisal-kernel

Входной язык системы ACL2 [14] является аппликативным и строго функциональным диалектом языка Common Lisp. Так как Cloud Sisal также является функциональным языком, то в работе [22] предложена трансляция выражений Cloud Sisal в выражения ACL2. Функция *sisal2acl2* реализует эту трансляцию. Определение этой функции описано в работе [22]. Рассмотрим краткое описание этой трансляции.

3.1. Трансляция базовых конструкций языка Cloud-Sisal-kernel на язык ACL

Функция *sisal2acl2* конвертирует массивы языка Cloud-Sisal-kernel в списки ACL2. Две операции используются для обработки индексируемых последовательностей. Рассмотрим функции *nth* и *update-nth*, которые реализуют эти операции. Если i является индексом и l является списком, тогда (*nth i l*) является значением i-го элемента из списка l. Если выражение *expr* является выражением ACL2, тогда (*update-nth i expr l*) является новым списком, который совпадает со списком l за исключением i-го элемента, чьим значением является *expr*.

Функция *sisal2acl2* транслирует триплеты Cloud-Sisal-kernel в применение 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*.

Мы используем специальные конструкции, предоставляемые библиотекой *fty* в системе ACL2, для задания новых типов. Если *e* – имя нового типа, тогда генерируется макрос *make-e* с помощью библиотеки *fty*. Этот макрос является конструктором для типа *e*. Мы используем этот подход для создания структуры *loop-expr* в главе 4.

Также эти конструкции позволяют задать специальные структуры для моделирования контекста цикла. Тип структуры *environment_id* генерируется для моделирования контекста. Поля этой структуры соответствуют переменным контекста. Мы генерируем функцию *create_environment_id* для упрощения создания объектов типа *environment_id*. Функции *context_variables*, *context2vector* и *context2string* позволяют получать информацию о переменных контекста от компилятора CPPS [15, 16, 21] и конвертировать список таких переменных в строковое представление.

3.2. Трансляция циклов Cloud-Sisal-kernel на язык ACL

Аксиоматическая семантика циклических выражений Cloud-Sisal-kernel основана на символическом методе верификации финитных итераций [12]. Следовательно, функция *sisal2acl2* транслирует цикл в применение функции *rep_id*, где *id* является уникальным идентификатором.

Отметим, что в работе [22] не была задана аксиоматическая семантика для циклических выражений с условиями завершения.

Циклы языка Cloud-Sisal-kernel задают итерации над кортежами значений переменных из заголовка цикла. Эти кортежи получаются декартовым произведением триплетов из заголовка циклов. Следовательно, трансляция определена с помощью следующего применения функции *range2acl2*:

$$
\begin{aligned}
&sisal2acl2(loop\_e) \;=\; (rep\_id \\
&(reverse\ range2acl2(triplet_1\ cross\ \ldots\ triplet_{n-1}\ cross\ triplet_n)) \\
&\quad (create\_environment\_id\ context2string\ context2vector(\\
&\qquad context\_variables(expr) \setminus \{var_1,\ var_2,\ \ldots\ var_{n-1},\ var_n\}))),
\end{aligned}
$$

где *id* является уникальным идентификатором. 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*.

Генерация определения функции *rep_id* также описана в работе [22]. Функция *sisal2acl2* реализует эту генерацию как трансляцию тела цикла в применение конструкции $b*$ языка ACL2. Рассмотрим общую форму этой конструкции: $(b*\ (\ldots(var\ expr)\ldots)\ result\_expr)$, где выражение $(var\ expr)$ обозначает связывание переменной *var* со значением выражения *expr*, которое может зависеть от ранее связанных переменных. Значением блока $b*$ является значение выражения *result_expr*, которое также может зависеть от связываемых переменных. Отметим, что значение переменных из заголовка цикла задано в том кортеже из декартового произведения, над которым исполняется текущая итерация. Рассмотренные конструкции позволяют создать в блоке $b*$ переменные, соответствующие переменным из заголовка цикла, и связать их значения с соответствующими значениями из кортежа, над которым исполняется текущая итерация. Исполнение конструкций, завершающих исполнение, моделируется истинностью условия *when*, которое может быть использовано как условие связываний.

Отметим, что переменные заголовка цикла языка Cloud-Sisal-kernel не зависят друг от друга. Поэтому, мы не используем такую возможность блока $b*$, как зависимость переменных от ранее определенных переменных. Мы используем блок $b*$, чтобы задать переменные, соответствующие переменным контекста, и переменные, соответствующие переменным заголовка цикла. Эти переменные не зависят друг от друга. Так как в языке системы ACL2 отсутствуют замыкания, то задание в блоке $b*$ таких переменных позволяет моделировать возможности замыкания. Введение таких переменных позволяет записать внутри блока $b*$ редуцируемое выражение, зависящее от переменных контекста цикла и от переменных заголовка цикла.

Функция *sisal2acl2* использует транслятор *reduct2acl2* для генерации применения редуцирующей функции как *result_expr* конструкции $b*$. Функция *reduct2acl2* реализует трансляцию редукций. Эта функция принимает три аргумента: имя редукции, редуцируемое выражение на текущей итерации $expr_1$ и значение цикла после предыдущей итерации ($expr_2$). Мы также используем вспомогательную функцию *reduction_init*, которая принимает в качестве аргумента имя редукции и возвращает значение редукции по умолчанию. Эти функции позволяют реализовать редукцию в теле функции *rep*. Рассмотрим список некоторых редуцирующих функций, полученных в результате применения транслятора *reduct2acl2*:

- $reduct2acl2(array\ of, expr_1, expr_2) = (cons\ expr_1\ expr_2)$
- $reduct2acl2(sum\ of, expr_1, expr_2) = (+\ expr_1\ expr_2)$
- $reduct2acl2(product\ of, expr_1, expr_2) = (*\ expr_1\ expr_2)$

Редуцирующая функция в теле функции *rep_id* применяется к редуцируемому выражению и результату рекурсивного вызова функции *rep_id*.

3.3. Трансляция выражений замещения элементов массивов на язык ACL

Семантика выражений замещения элементов массивов также основана на символическом методе верификации финитных итераций. Эти выражения транслируются в применение рекурсивных функций *update_elements_id*, где *id* является уникальным идентификатором. Рассмотрим трансляцию этих выражений с помощью функции *sisal2acl2*:

$sisal2acl2(array\_rep\_expr) =$
$(update\_elements\_id$
$(reverse\ range2acl2(triplet_1\ cross\ \ldots\ cross\ \ldots\ triplet_{n-1}\ cross\ triplet_n))$
$(create\_environment\_id$
$context2string($
$context2vector($
$context\_variables(expr) \setminus \{var_1,\ var_2,\ \ldots\ var_{n-1},\ var_n\})))$
$source\_array),$

Отметим, что применение функции $update\_elements\_id$ похоже на применение функции rep_id в случае циклических выражений. Определение функции $update\_elements\_id$ также схоже с определением функции rep_id за исключением результирующего выражения в блоке $b*$ с применениями выражений $update\text{-}nth$ вместо редуцирующей функции.

3.4. Аксиоматическая семантика языка Cloud-Sisal-kernel

Для задания аксиоматической семантики языка Cloud-Sisal-kernel используется метод слабейшего предусловия [7]. Данный метод, как и символический метод верификации финитных итераций, основан на замене переменных их значениями, вычисленными символически. 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 $P \longrightarrow wp(S, Q)$ следует частичная корректность тройки $\{P\}\ S\ \{Q\}$. Данное свойство позволяет использовать метод слабейшего предусловия для генерации УК.

Аксиоматическая семантика выражений языка Sisal описана в работе [22]. Для задания аксиоматической семантики язык спецификаций был расширен термом $result$. Данный терм можно использовать в постусловии Sisal-выражения для обозначения значения этого выражения. Пусть $R(y \longleftarrow expr)$ обозначает одновременную замену в R всех свободных вхождений переменной y на на $expr$. Если $expr$ является выражением языка Sisal, тогда

$$\mathbf{wp(expr, Q) = Q(result \longleftarrow sisal2acl2(expr))}$$

Так как слабейшее предусловие Sisal-выражений основано на применении транслятора $sisal2acl2$, то аксиоматическая семантика языка Cloud-Sisal-kernel основана на трансляционной семантике. Отметим, что в работе [22] не была задана аксиоматическая семантика для циклических выражений с условиями завершения.

4. Расширение языка C-kernel циклическими выражениями языка Cloud-Sisal-kernel

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*. 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 написание кода 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 синтаксисом *array_rep_expr*. 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. 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* [5] 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.

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. Все инструкции, содер-

жащие 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:

$$\{P\}\ \mathbf{A};\ \{Q(var\ \leftarrow c2acl2(low\ ..hight\ ..step),\ MD \leftarrow upd(MD, var.vector, c2acl2(low\ ..high\ ..step).vector))\}$$

$$\{P\}\ \mathbf{A}; \mathbf{var}\ =\ \mathbf{low}\ ..\mathbf{high}\ ..\mathbf{step};\ \{Q\}$$

где A являются инструкциями программы до рассматриваемого присваивания. Мы используем обратное прослеживание (метод слабейшего предусловия [7]): мы движемся от конца программы к ее началу и элиминируем самый правый оператор (на верхнем уровне), применяя соответствующее правило вывода смешанной аксиоматической семантики [1] 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:

$$\{P\}\ \mathbf{A};\ \{Q(var\ \leftarrow c2acl2(loop\_e),\ MD \leftarrow upd(MD, var.vector, c2acl2(loop\_e).vector))\}$$

$$\{P\}\ \mathbf{A};\ \mathbf{var}\ =\ \mathbf{loop\_e}\ \{Q\}$$

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:

$$\begin{aligned}
&c2acl2(loop_e)\ =\ (make\text{-}loop_expr\\
&:\ scalar\ (rep_id\\
&\quad (reverse\ range2acl2(triplet_1\ cross\ \ldots\ cross\ triplet_n))\\
&\quad (create_environment_id\ context2string(context2vector(\\
&\qquad context_variables(expr) \setminus \{var_1,\ var_2,\ \ldots\ var_{n-1},\ var_n\}))))\\
&:\ vector\ nil)
\end{aligned}$$

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:

$$\begin{aligned}
&(defun\ rep_id\ (range_tuples\ environment)\\
&\ (b*\ ((when\ (endp\ range_tuples))\ reduction_init(reduction))\\
&\quad (tuple\ (car\ range_tuples))\\
&\quad (var_1\ (car\ tuple))
\end{aligned}$$

```
   ...
   (var_n (car (cdr (cdr ... (cdr tuple) ...))))
   (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 [22] является применение транслятора *c2acl2* [5] 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 [2] 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 [5] 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. Отметим, что такой алгоритм отсутствует для циклов с условиями завершения в языке Cloud-Sisal-kernel. Рассмотрим такую генерацию определения функции *rep_id*:

```
(defun rep_id (range_tuples environment)
 (b *
  (when (endp range_tuples))(update : loop-break nil reduction_init(reduction)))
  (tuple (car range_tuples))
  (var_1 (car tuple))
  ...
  (var_n (car (cdr (cdr ... (cdr tuple) ...))))
  (previous_iter_result (rep_id (cdr range_tuples) environment))
  ((when (not previous_iter_result.loop-break)) previous_iter_result)
  ((when (not c2acl2(condition))) (update : loop-break t previous_iter_result)))
 (update : loop-break nil
  reduct2acl2(reduction, c2acl2(expr), previous_iter_result))))
```

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:

$$\{P\}\ \mathbf{A};\ \{Q(source\_array \leftarrow c2acl2(array\_rep\_expr),\ MD \leftarrow upd(MD, source\_array, c2acl2(array\_rep\_expr)))\}$$

$$\{P\}\ \mathbf{A};\ \mathbf{array\_rep\_expr};\ \{Q\}$$

где A определяется 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)
   (b* ((when (endp range_tuples)) source_array)
       (tuple (car range_tuples))
       (var_1 (car tuple))
       ...
       (var_n (car (cdr (cdr ... (cdr tuple) ... )))))
     (update-nth var_1
     ...
     (update-nth var_{n-1}
     (update-nth var_n c2acl2(expr)
       (nth var_{n-1}
       ...
       (nth var_1
         (update_elements_id(cdr range_tuples) environment
           source_array))) ...))))
```

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 [2] 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. Эксперименты

Опишем эксперименты, демонстрирующие применение представленной семантики для автоматической дедуктивной верификации C-Sisal-kernel программ.

5.1. Произведение элементов матрицы

Автоматическая дедуктивная верификация суммирования элементов матрицы описана в работе [22]. Данная программа была задана на языке Cloud-Sisal-kernel. Рассмотрим автоматическую верификацию схожей программы, заданной на языке C-Sisal-kernel. Эта программа вычисляет произведение элементов целочисленной матрицы:

```
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 нами в системе ACL2 как список целочисленных списков. Длины всех вложенных списков равны. Предикат *integer-matrixp* проверяет, соответствует ли его аргумент этой структуре.

Постусловием является формула `(= prod (product-matrix n m a))`. Функция *product-matrix* задана нами на языке ACL. 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 [1] и метод метагенерации УК [2] 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 функциях *integer-matrixp* и *product-matrix*. Следовательно, функция `product_matrix_elements` соответствует спецификациям.

5.2. Изменение знака первого отрицательного элемента массива на противоположный

Рассмотрим программу `negate_first` из набора задач по верификации [26]. 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 [1] и метод метагенерации УК [2] 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 автоматизации доказательства УК программ, постусловием которых является разбор случаев [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`. Эти леммы были добавлены в теорию предметной области. Система ACL2 автоматически доказала оба УК, используя индукцию по *n* и эти леммы. Следовательно, функция `negate_first` соответствует спецификациям.

Заключение

В данной статье представлены следующие основные результаты:

- Новый язык программирования C-Sisal-kernel:
  - синтаксис языка C-Sisal-kernel;
  - семантика языка C-Sisal-kernel.
- Эксперименты по автоматической верификации C-Sisal-kernel программ:

– произведение элементов матрицы;
– изменение знака первого отрицательного элемента массива на противоположный.

В данной статье также представлен комплексный подход к автоматизации дедуктивной верификации в случае финитных итераций над структурами данных. Данный подход представляет собой композицию символического метода верификации финитных итераций, метода метагенерации УК и метода смешанной аксиоматической семантики. Символический метод верификации финитных итераций позволяет решить проблему инвариантов для циклов определенного вида. Метод метагенерации УК позволяет упростить добавление новых правил вывода в систему верификации. Метод смешанной аксиоматической семантики позволяет получить более простые УК в некоторых случаях.

Мы планируем расширить язык C-Sisal-kernel такими циклами Cloud Sisal, как выражения конструирования массива. Для решения этой задачи мы планируем применить представленный комплексный подход к автоматизации дедуктивной верификации.

Также мы планируем изменить реализацию синтаксического анализа выражений *upd* из правил вывода. Этот анализ является частью метагенератора. Применение анализа к выражению *upd* из правила вывода не зависит от выражений *upd* в других правилах вывода. Следовательно, мы можем реализовать такой анализ, используя Sisal-итерации над строками, соответствующими правилам вывода. Автоматическое распараллеливание этой части метагенератора будет основано на использовании языка C-Sisal-kernel. Такая реализация может позволить выполнить эксперименты по применению системы верификации к этой части собственного исходного кода. Отметим, что формальная верификация генераторов УК является актуальной задачей [2, 43]. Планируемое самоприменение системы C-lightVer будет шагом на пути к решению этой задачи.

References

[1] I. V. Maryasov, V. A. Nepomniaschy, A. V. Promsky, and D. A. Kondratyev, “Automatic C Program Verification Based on Mixed Axiomatic Semantics”, *Automatic Control and Computer Sciences*, vol. 48, no. 7, pp. 407–414, 2014.

[2] D. A. Kondratyev and A. V. Promsky, “Developing a self-applicable verification system. Theory and practice”, *Automatic Control and Computer Sciences*, vol. 49, no. 7, pp. 445–452, 2015.

[3] D. Kondratyev, “Implementing the Symbolic Method of Verification in the C-Light Project”, in *Perspectives of System Informatics*, ser. LNCS, vol. 10742, Springer, 2018, pp. 227–240.

[4] D. A. Kondratyev, I. V. Maryasov, and V. A. Nepomniaschy, “The Automation of C Program Verification by the Symbolic Method of Loop Invariant Elimination”, *Automatic Control and Computer Sciences*, vol. 53, no. 7, pp. 653–662, 2019.

[5] D. A. Kondratyev and A. V. Promsky, “The Complex Approach of the C-lightVer System to the Automated Error Localization in C-Programs”, *Automatic Control and Computer Sciences*, vol. 54, no. 7, pp. 728–739, 2020.

[6] C. A. R. Hoare, “An axiomatic basis for computer programming”, *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[7] K. R. Apt and E.-R. Olderog, “Fifty years of Hoare’s logic”, *Formal Aspects of Computing*, vol. 31, no. 6, pp. 751–807, 2019.

[8] R. Hähnle and M. Huisman, “Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools”, in *Computing and Software Science*, ser. LNCS, vol. 10000, Springer, 2019, pp. 345–373.

[9] K. R. Apt and E.-R. Olderog, “Assessing the Success and Impact of Hoare’s Logic”, in *Theories of Programming: The Life and Works of Tony Hoare*, 2021, pp. 41–76.

[10] V. A. Nepomniaschy, I. S. Anureev, I. N. Mikhailov, and A. V. Promskii, "Towards verification of C programs. C-light language and its formal semantics", *Programming and Computer Software*, vol. 28, no. 6, pp. 314–323, 2002.

[11] V. A. Nepomniaschy, I. S. Anureev, and A. V. Promskii, "Towards Verification of C Programs: Axiomatic Semantics of the C-kernel Language", *Programming and Computer Software*, vol. 29, no. 6, pp. 338–350, 2003.

[12] V. A. Nepomniaschy, "Symbolic method of verification of definite iterations over altered data structures", *Programming and Computer Software*, vol. 31, no. 1, pp. 1–9, 2005.

[13] M. Moriconi and R. L. Schwartz, "Automatic construction of verification condition generators from hoare logics", in *International Colloquium on Automata, Languages, and Programming*, ser. LNCS, vol. 115, Springer, 1981, pp. 363–377.

[14] J. S. Moore, "Milestones from the Pure Lisp Theorem Prover to ACL2", *Formal Aspects of Computing*, vol. 31, no. 6, pp. 699–732, 2019.

[15] V. Kasyanov and E. Kasyanova, "Methods and System for Cloud Parallel Programming", in *Proceedings of the 21st International Conference on Enterprise Information Systems*, vol. 1, 2019, pp. 623–629.

[16] V. N. Kasyanov and A. P. Stasenko, "Sisal 3.2 Language Structure Decomposition", in *Proceedings of the European Computing Conference*, ser. LNEE, vol. 28, Springer, 2009, pp. 533–543.

[17] A. Stasenko, "Sisal 3.2 Language Features Overview", in *International Conference on Parallel Computing Technologies*, ser. LNCS, vol. 6873, Springer, 2011, pp. 110–124.

[18] V. Kasyanov, "Sisal 3.2: functional language for scientific parallel programming", *Enterprise Information Systems*, vol. 7, no. 2, pp. 227–236, 2013.

[19] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the sisal language project", *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 349–366, 1990.

[20] J.-L. Gaudiot, T. DeBoni, J. Feo, W. Böhm, W. Najjar, and P. Miller, "The Sisal Project: Real World Functional Programming", in *Compiler Optimizations for Scalable Parallel Systems*, ser. LNCS, vol. 1808, Springer, 2001, pp. 45–72.

[21] K. Pyzhov and R. Idrisov, "Back-end translator for Sisal 3.1 compiler", *Bulletin of the Novosibirsk Computing Center*, no. 35, pp. 101–119, 2013.

[22] D. A. Kondratyev and A. V. Promsky, "Towards verification of scientific and engineering programs. The CPPS project", *Journal of Computational Technologies*, vol. 25, no. 5, pp. 91–106, 2020.

[23] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, vol. 6, 2004.

[24] M. Kaufmann and J. S. Moore, "Iteration in ACL2", in *Proceedings of the Sixteenth International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, vol. 327, 2020, pp. 16–31.

[25] S. Blom, S. Darabi, M. Huisman, and M. Safari, "Correct program parallelisations", *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 5, pp. 741–763, 2021.

[26] B. Jacobs, J. Kiniry, and M. Warnier, "Java Program Verification Challenges", in *Formal Methods for Components and Objects*, ser. LNCS, vol. 2852, Springer, 2003, pp. 202–219.

[27] D. R. Cok, "Reasoning about Functional Programming in Java and C++", in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018, pp. 37–39.

[28] D. R. Cok and S. Tasiran, "Practical Methods for Reasoning About Java 8's Functional Programming Features", in *Verified Software: Theories, Tools, and Experiments*, ser. LNCS, vol. 11294, Springer, 2018, pp. 267–278.

[29] *ISO/IEC 14882:2020: Programming language C++*. ISO/IEC, 2020.

[30] *ISO/IEC 9899:2011: Programming language C*. ISO/IEC, 2011.

[31] R. Krebbers and F. Wiedijk, "A Typed C11 Semantics for Interactive Theorem Proving", in *Proceedings of the 2015 Conference on Certified Programs and Proofs*, 2015, pp. 15–27.

[32] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg, "RefinedC: automating the foundational verification of C code with refined ownership types", in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 158–174.

[33] M. O. Myreen and M. J. C. Gordon, "Transforming Programs into Recursive Functions", *Electronic Notes in Theoretical Computer Science*, vol. 240, pp. 185–200, 2009.

[34] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter, "An overview of the Leon verification system: verification by translation to recursive functions", in *Proceedings of the 4th Workshop on Scala*, 2013, pp. 1–10.

[35] A. Humenberger, M. Jaroschek, and L. Kovács, "Invariant Generation for Multi-Path Loops with Polynomial Assignments", in *Verification, Model Checking, and Abstract Interpretation*, ser. LNCS, vol. 10747, Springer, 2018, pp. 226–246.

[36] S. Chakraborty, A. Gupta, and D. Unadkat, "Diffy: Inductive Reasoning of Array Programs Using Difference Invariants", in *Computer Aided Verification*, ser. LNCS, vol. 12760, Springer, 2021, pp. 911–935.

[37] T. Tuerk, "Local reasoning about while-loops", in *Proceedings of the Theory Workshop at VSTTE 2010*, 2010, pp. 29–39.

[38] A. Blanchard, F. Loulergue, and N. Kosmatov, "Towards Full Proof Automation in Frama-C Using Auto-active Verification", in *NASA Formal Methods*, ser. LNCS, vol. 11460, Springer, 2019, pp. 88–105.

[39] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams, "The dogged pursuit of bug-free C programs: the Frama-C software analysis platform", *Communications of the ACM*, vol. 64, no. 8, pp. 56–68, 2021.

[40] I. Attali, D. Caromel, and A. Wendelborn, "A Formal Semantics and an Interactive Environment for Sisal", in *Tools and Environments for Parallel and Distributed Systems*, ser. SOFT, vol. 2, Springer, 1996, pp. 229–256.

[41] D. Kondratyev and A. Promsky, "Proof Strategy for Automated Sisal Program Verification", in *Software Technology: Methods and Tools*, ser. LNCS, vol. 11771, Springer, 2019, pp. 113–120.

[42] B. Beckert, T. Bingmann, M. Kiefer, P. Sanders, M. Ulbrich, and A. Weigl, "Relational Equivalence Proofs Between Imperative and MapReduce Algorithms", in *Verified Software. Theories, Tools, and Experiments*, ser. LNCS, vol. 11294, Springer, 2018, pp. 248–266.

[43] G. Parthasarathy, P. Müller, and A. Summers, "Formally Validating a Practical Verification Condition Generator", in *Computer Aided Verification*, ser. LNCS, vol. 12760, Springer, 2021, pp. 704–727.