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

В. А. Непомнящий, И. С. Ануреев,
И. Н. Михайлов, А. В. Промский

НА ПУТИ К ВЕРИФИКАЦИИ С-ПРОГРАММ. ЧАСТЬ 3. ПЕРЕВОД ИЗ ЯЗЫКА C-LIGHT В ЯЗЫК C-LIGHT-KERNEL И ЕГО ФОРМАЛЬНОЕ ОБОСНОВАНИЕ

Препринт
97

Новосибирск 2002

Описаны правила перевода из языка C-light в язык C-light-kernel, являющиеся основой двухуровневой схемы верификации C-программ. Для языка C-light предложена модифицированная операционная семантика. Модификация позволяет упростить как описание семантики сложных конструкций языка C-light, так и доказательство непротиворечивости аксиоматической семантики языка C-light-kernel. Определено понятие семантического расширения и проведено формальное обоснование корректности перевода. Предполагается реализовать правила перевода в системе верификации программ.

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

V. A. Nepomniaschy, I. S. Anureev,
I. N. Michailov, A. V. Promsky

TOWARDS THE VERIFICATION OF C PROGRAMS. PART 3. TRANSLATION FROM C-LIGHT INTO C-LIGHT-KERNEL AND ITS FORMAL PROOF

Preprint
97

Novosibirsk 2002

Two-level scheme of C-program verification is based on rules for translation from C-light language to C-light-kernel. The translation rules are described in this paper. Operational semantics of C-light is modified in order to simplify both description of semantics of complex C-light constructs, and proving soundness of axiomatic C-light-kernel semantics. A notion of semantical extension is defined, and formal justification of correctness of this translation is given in this paper. We suggest to implement the translation rules in a program verification tool.

1. ВВЕДЕНИЕ

Формальная верификация программ — актуальное направление современного программирования. Особый интерес представляет верификация программ, написанных на распространенных языках системного программирования, таких как C и C++. Трудности верификации программ на языке C подробно обсуждались в [4–6, 9–11, 15]. Напомним, что основной проблемой является отсутствие формальной семантики для полного языка C, соответствующего стандарту ANSI [1–3]. Отметим, что формальная семантика для довольно представительного подмножества C была предложена в [12, 13].

Наш подход к верификации C-программ [3–5] состоит в следующем. Выделено представительное подмножество языка C, названное C-light. Сравнительный анализ работ, посвященных анализу и верификации C-программ, показал, что это одно из наиболее обширных подмножеств языка C. Оно содержит все множество операторов языка C при некоторых семантических ограничениях и большинство типов и операций. Отличительной чертой языка C-light является детерминированная семантика выражений. Вместо библиотечных функций для работы с динамической памятью используются операции new и delete языка C++. В качестве формального определения языка C-light [5] была разработана его полная структурная операционная семантика [17].

При верификации программ вместо громоздкой операционной семантики обычно используется аксиоматическая семантика в стиле Хоара. Эта семантика более высокого уровня, что позволяет существенно упростить доказательство корректности программ. Непосредственное определение аксиоматической семантики для C-light будет весьма громоздким, что усложнит верификацию. Выходом стало применение двухуровневой схемы верификации C-light-программ. В соответствии с ней в языке C-light выделяется ядро, «хорошее» с точки зрения аксиоматической семантики, в которое транслируются исходные программы. Для этого ядра, названного C-light-kernel, была разработана аксиоматическая семантика и доказана ее непротиворечивость относительно операционной [6].

Двухуровневая схема верификации предусматривает также формальное определение правил перевода из языка C-light в язык C-light-kernel и доказательство их корректности. Понятно, что эта задача является трудной. Отметим, что системы эквивалентных преобразований, упрощающих выражения языка C с побочными эффектами, изучались в [9].

Однако, как справедливо отмечалось в [16], доказательство корректности преобразований не описано. В данной работе предлагается решение задачи формального определения правил перевода и обоснования их корректности.

В разд. 2 рассмотрена модифицированная операционная семантика языка C-light. Модификация позволила более адекватно описать поведение сложных конструкций и упростить доказательство корректности аксиоматической семантики языка C-light-kernel, поскольку она также определяется в терминах новой операционной семантики. В частности, явное моделирование абстрактной памяти позволило избавиться от громоздких типов ссылочных классов для указателей [14].

Разд. 3 посвящен формальному определению системы правил перевода из языка C-light в язык C-light-kernel. Правила разбиты на группы в соответствии со стратегией их применения. Там же рассмотрены некоторые свойства правил перевода.

В разд. 4 проводится формальное доказательство корректности системы перевода. Для этого предложено понятие семантического расширения. В отличие от обычного отношения функциональной эквивалентности, это отношение является предпорядком, т.е. нет свойства симметричности, хотя сохраняются свойства рефлексивности и транзитивности. Доказательство корректности системы разбито на три этапа: доказательство корректности правил, т.е. сохранения семантики преобразуемой программы, завершимости процесса перевода и теоремы о нормальной форме. Следствием корректности системы перевода является важное свойство сохранения частичной корректности аннотированных программ.

Эта работа частично поддержана грантами РФФИ 00-01-00909 и 01-01-06141.

2. ОПЕРАЦИОННАЯ СЕМАНТИКА ЯЗЫКА C-LIGHT

Операционная семантика — это отображение, которое сопоставляет каждому элементу из синтаксической области определения значение или интерпретацию, т. е. элемент семантической области. Тем самым описывается смысл каждой конструкции языка. Описание смысла должно быть полностью формальным и логически непротиворечивым. Для этого определяется абстрактная вычислительная машина, которая интерпретирует синтаксис языка. Фактически при этом моделируется реальное исполнение программы.

В п. 2.1 дается краткий обзор языка C-light, в котором отражены последние изменения и ограничения. Далее в п. 2.2 определяется понятие состояния абстрактной вычислительной машины языка C-light. В п. 2.3 рассмотрены аксиомы и правила определения типов выражений и значений. Конфигурации вычислительной машины и процесс интерпретации программ определены в п. 2.4. Семантика выражений, деклараций и операторов рассматривается в пп. 2.5-2.7.

2.1. Краткий обзор языка C-light

Предыдущие версии языков C-light и C-light-kernel были определены в [5, 6], где подробно рассмотрены синтаксис и семантика. Напомним, что вместе с языком ANSI C они образуют следующую неформальную последовательность:

$$\text{ANSI C} \supset \text{C-light} \supset \text{C-light-kernel}.$$

Включение означает как синтаксические, так и семантические ограничения. Рассмотрим структуру языка C-light с учетом последних модификаций[1].

Типы. Допустимыми типами языка C-light (и, следовательно, языка спецификации программ) являются следующие:

- **Базовые типы**
  - целочисленные: bool, wchar_t
    i ::= char, int, short [int], long [int]
    τ ::= signed i, unsigned i
    enum
  - вещественные: float, double, long double
  - пустой: void
- **Составные типы**
  - указатели: $T*$
  - массивы: T[n]
  - структуры: struct$(T_1\ v_1; ...; T_n\ v_n)$
  - функции: $T_1 \times ... \times T_n \to T$

Новые базовые типы языка C, предлагаемые последним стандартом [3], не поддерживаются. Также отметим, что имя логического типа не _Bool, а bool, как в C++.

[1] Обзор языка C-light-kernel дается в п. 3.1.

При этом на допустимые типы накладываются следующие ограничения:

1) тип char всегда имеет знак;
2) элементы перечислимых типов являются объектами типа signed int;
3) значениями указателей являются неинтерпретируемые константы;
4) запрещены указатели на функции;
5) неполные типы массивов, т.е. без указания размера, разрешены только как типы аргументов функций;
6) в структурах запрещены битовые поля;
7) функции стандартной библиотеки поддерживаются только при наличии исходных файлов;
8) запрещены функции с переменным числом аргументов;
9) запрещены абстрактные декларации аргументов и аргументы по умолчанию.

Декларации. Рассмотрим основные отличия деклараций в языке C-light от деклараций языка ANSI C.

Неокончательные определения (tentative definitions) из языка ANSI C запрещены.

Запрещены абстрактные декларации аргументов и аргументы по умолчанию. Пустой список аргументов не разрешен — требуется void.

Спецификация декларации перечисляет свойства объекта, которые нужны в основном для оптимизации кода. Операционная среда C-light существенно проще реальной операционной среды C. Поэтому запрещены все спецификаторы и модификаторы типов, кроме спецификаторов класса памяти, спецификаторов наличия знака и спецификаторов размера для скалярных типов.

По сравнению с [5] появились два новых ограничения. Согласно первому ограничению правила разрешения имен распространяются на спецификации. Отсюда следует, что либо все имена в программе уникальны, либо есть переобъявления, но в спецификации конкретное имя обозначает один объект, т.е. это не запрет на локальные переименовывания вообще, а правило написания *аннотированных программ*. Согласно второму ограничению имена *всех* статических объектов в программе уникальны, т.е. не только глобальные объекты, но и имена переменных, объявленных в телах функций с ключевым словом static.

Выражения. Главными отличиями языка C-light от C являются наличие детерминизма вычисления и запрет побитовых операций. Подробнее синтаксис выражений рассматривается при определении статической семантики.

В C-light все выражения равноправны. И сложное адресное выражение, и отдельный идентификатор переменной обрабатываются по одним и тем же правилам. Выражение общего вида есть список выражений присваивания, разделенных запятыми. Эти выражения будут вычисляться строго слева направо, результат самого правого из них будет результатом всего выражения. Выражение присваивания может либо содержать операции присваивания, либо быть простым rvalue.

В C-light предусмотрены все операции присваивания C, кроме составных присваиваний, содержащих побитовые операции.

Для приведения типов используется синтаксис языка C. Допустимы только безопасные приведения типов (главным образом для скалярных типов). Для указателей введено дополнительное ограничение — можно делать только приведение от типа void* к T*, где T — любой стандартный тип или тип пользователя.

Для выделения и освобождения динамической памяти используются операции new и delete из языка C++.

Операторы. Оператором (инструкцией) языка C-light является любой оператор языка C, в том числе определение переменной или типа данных «на лету» (как в C++).

Пустой оператор тоже является оператором вычисления выражения.

Типом условных выражений в циклах и условных операторах может быть либо скалярный тип, либо bool. Тип выражения в операторе return должен приводиться по умолчанию к типу, возвращаемому функцией, в которой содержится этот оператор.

Мы считаем, что в условном операторе if всегда есть ветвь else, возможно пустая. Все case-метки в операторе switch должны находиться на одном уровне вложенности.

Запрещено передавать управление по goto внутрь любого блока из охватывающего его блока или из одного блока в другой, не пересекающийся с ним. Однако можно передать управление из вложенного блока в охватывающий. Как и в C++, запрещено передавать управление по goto в обход инициализации.

Исходная программа. Весь исходный текст программы есть последовательность внешних деклараций. На внешнем уровне можно объявить тип (typedef-декларации), функцию (прототип или определение), структуру и данные. Кроме обычных конструкций C на верхнем уровне могут присутствовать и аннотации, подробно рассмотренные в [5].

В языке C-light препроцессор отсутствует, поскольку исходная программа препроцессируется до начала ее верификации. Также, по сравнению с [16], не предусмотрена никакая предварительная компиляция.

В процессе верификации модульность не поддерживается. Поэтому с точки зрения семантики любая исходная программа состоит из одного файла. Например, библиотеки пользователя или стандартная библиотека C верифицируются отдельно от программы, которая использует их.

2.2. Состояния абстрактной машины языка C-light

Операционная семантика рассматривает процесс исполнения программы в терминах изменений состояний некоторой вычислительной машины и, возможно, в терминах взаимодействия программы с внешним окружением. Уровень детализации состояния может быть различным, однако, в соответствии со стандартом языка C, вычислительная машина должна быть абстрактной и не связанной с какой-либо конкретной архитектурой. Хотя память машины моделируется с высоким уровнем абстрактности, для вычисления выражений выбран конкретный подход [5, 6]. Это позволяет существенно упростить состояния машины по сравнению с [16]. Тем не менее, мы придерживаемся стандарта в отношении многих других неспецифицированных аспектов языка C. Мы также не рассматриваем стандартную библиотеку языка C и вызовы системных функций при отсутствии их исходных текстов, поэтому взаимодействие с внешним окружением не моделируется.

Семантическая область. Определение операционной семантики начиняется с того, что для каждого типа T фиксируется множество значений, называемое *носителем* типа T и обозначаемое D_T. Поскольку мы не ориентируем семантику на конкретную архитектуру, то границы носителей базовых типов задаются символическими константами. Заметим, что одним из методов интерпретации этих констант может быть инициализация их значениями из стандартного файла `limits.h`.

- $D_{\mathbf{bool}} = \{\text{FALSE}, \text{TRUE}\}$;

- $D_{\textbf{unsigned char}} = \{0 \ldots \text{MAX\_UNSIGNED\_CHAR}\}$;
- $D_{\textbf{signed char}} = \{\text{MIN\_SIGNED\_CHAR} \ldots \text{MAX\_SIGNED\_CHAR}\}$;
- $D_{\textbf{wchar\_t}} = D_{\textbf{unsigned short}}$;
- $D_{\textbf{enum}} = D_{\textbf{signed int}}$ для любого перечисления;
- $D_{\textbf{void}} = \emptyset$;
- $D_{T*} = D_{\textbf{unsigned int}}$ для любого типа T;
- $D_{T[\textbf{n}]} = D_T^n$ (декартова степень n) для любого непустого и нефункционального типа T;
- $D_{\textbf{struct}(T_1\ v_1;\ \ldots;\ T_n\ v_n)} = D_{T_1} \times \ldots \times D_{T_n}$;
- $D_{T_1 \times \ldots \times T_n \to T} = D_{T_1} \times \ldots \times D_{T_n} \to D_T$, т. е. множество всех функций из декартова произведения множеств $D_{T_1}, \ldots, D_{T_n}$ во множество D_T.

Отметим, что по сравнению с [5] отсутствуют типы ссылочных классов. Явное моделирование памяти абстрактной машины позволяет избавиться от этих сложных типов, для которых приходится использовать сложную нотацию.

Семантическая область $\boldsymbol{D}$ определяется как объединение по всем типам:

$$\boldsymbol{D} = \bigcup_T D_T.$$

Для любой константы типа T фиксируем значение в носителе D_T и говорим, что константа *обозначает* это значение. Считаем, что каждая константа базового типа (а также типов «массив» и «структура») обозначает саму себя. В свою очередь, константы типов «функций» обозначают соответствующие функции.

В отличие от констант значения переменных не фиксированы и определяются через состояния абстрактной вычислительной машины. *Состояние* — это в простейшем случае отображение, которое присваивает любой переменной типа T значение в области D_T. Очевидно, что такое определение состояния слишком слабое для языков C и C-light. Такой семантики достаточно для ограниченных модельных языков, в частности для небольшого подмножества стандартного Паскаля [7, 8]. В нашем случае помимо отображения содержимого абстрактной памяти в состоянии будут дополнительные компоненты. Степень абстрактности вычислительной машины определяется, в первую очередь, целями, поставленными при верификации.

Состояние абстрактной машины. По определению состояние абстрактной вычислительной машины языка C-light состоит из следую-

щих компонент.

1) IDs — семейство множеств имен существующих объектов;
2) Adrs — семейство множеств адресов объектов программы;
3) MeM — семейство отображений из IDs в Adrs;
4) MD — отображение всех модифицируемых объектов и констант программы во множество их потенциальных значений;
5) Γ — информация о типах объектов;
6) Σ — информация о структурах;
7) TD — информация о синонимах типов (typedef-декларации);
8) Val — специальная ячейка, в которой хранится значение последнего вычисленного (под)выражения вместе с его типом;
9) GLF — флаг, определяющий вложенность;
10) BC — кортеж из шести элементов — «история» работы некоторого блока.

По сравнению с предыдущим определением состояния [5], изменилось не только число компонент, но и структура некоторых из них. Рассмотрим компоненты подробнее.

1. IDs — семейство множеств имен существующих объектов. Отличительной чертой языка C является то, что программист неявно участвует в создании и уничтожении объектов. Различаются объекты с глобальными и локальными областями определенности, с глобальным и локальным временем жизни. Ранее существовало только глобальное множество всех объектов, и роль декларации сводилась только к связыванию имени с типом и возможной инициализации. Сейчас же IDs — это динамически изменяемый объект, для которого определены операции расширения и сужения именами создаваемых и уничтожаемых объектов в виде обычных теоретико-множественных операций объединения и разности. Только расширения и сужения недостаточно для согласования с правилами разрешения областей определенности и видимости языка C. Поэтому IDs — это семейство множеств, каждый элемент которого соответствует отдельной области видимости:

$$\mathrm{IDs} = \{\mathrm{ID}^i \mid \mathrm{ID}^i\text{– множество идентификаторов, } i \geq 0\}.$$

Тем самым возможна ситуация, когда каждая часть ID^i состоит из одного и того же идентификатора, но это все имена разных объектов. Индекс 0 всегда соответствует глобальной области определенности (файл), индекс i соответствует i-му уровню вложенности в теле текущей функции. Этот подход позволяет, во-первых, разрешить конфликты имен

в программе и при этом не вводить процесс частичной компиляции и таблицы имен. Во-вторых, просто моделируется сам процесс создания и уничтожения объектов. Вход в некоторый блок означает добавление новой (пока еще пустой) части ID^i с индексом, соответствующим уровню вложенности данного блока, а при выходе из блока эта часть удаляется. Процесс создания объектов в этом блоке моделируется расширением части ID^i.

2. Adrs — семейство множеств адресов объектов программы:

$$\mathrm{Adrs} = \{\mathrm{Adr}^i \mid \mathrm{Adr}^i\text{– множество адресов, } i \geq 0\}.$$

При этом есть прямое соответствие: Adr^i — это адреса объектов из ID^i. Однако, несмотря на внешнюю простоту, возникает серьезная проблема.

Каким образом можно определить адрес объекта в семантике? Очевидно есть некоторое соответствие адресов и элементов типа unsigned int. Целые числа образуют вполне упорядоченное множество. Но сравнить два адреса, даже в полностью формальной семантике стандарта языка C, невозможно. Стандарт никак не определяет раскладку объектов в памяти, и разные реализации делают это по-разному. Нет даже гарантии, что два последовательно объявленных объекта и в памяти займут соседние ячейки. Наконец особенности архитектуры могут повлиять на представление адресов (плоская и сегментированная модель памяти).

Поэтому все Adr^i определяются как множества неинтерпретируемых констант с равенством. Любые две константы, различающиеся синтаксически, будут различаться и значениями.

Заметим, что адреса динамически создаваемых объектов не могут храниться в Adr^i, при $i > 0$, поскольку выход из блока, в котором был создан такой объект, не означает его уничтожения. Поэтому эти адреса помещаются в Adr^0. Хотя мы избегаем прямых аналогий с конкретными архитектурами, это может означать следующее: Adr^0 соответствует сегменту данных и «куче», все остальное — это стек.

3. Ранее отмечалось, что для простого языка программирования бывает достаточно определить состояние как отображение имен переменных в носители типов. Значение переменной x в данном состоянии σ — это просто $\sigma(x)$. Этот подход годится для ассоциативной архитектуры, но неприемлим, если в языке есть указатели, которым можно присваивать адреса других объектов. Поэтому в данной семантике значение

объекта определяется не из его имени, а берется как содержимое памяти по адресу объекта. Для этого используется третья компонента MeM — отображение, устанавливающее соответствие первых двух компонент (MemoryManagement). Поскольку Ids и Adrs — семейства, то и MeM — это семейство функций:

$$\mathrm{MeM} = \{\mathrm{MeM}^i \mid \mathrm{MeM}^i : \mathrm{ID}^i \to \mathrm{Adr}^i,\ i \geq 0\}.$$

Так как в Adr^0 хранятся адреса динамических объектов, MeM^i — биекция только при $i > 0$. При $i = 0$ это только взаимно-однозначное отображение.

4. Итак, определив адрес объекта, можно узнать значение, находящееся по этому адресу. Компонента MD — это отображение всех модифицируемых объектов и констант программы во множество их потенциальных значений (MemoryDump):

$$\mathrm{MD} : \mathrm{Adrs} \to D.$$

Очевидно, что это не всюду определенное отображение, так как функция — это тоже объект.

Важно отметить, что, хотя мы и вводим адреса, побайтная (а тем более побитная) раскладка значений в памяти не моделируется. Это связано с особенностями стандарта языка C и возможными существенными различиями конкретных архитектур. Язык C не специфицирует ни размеры значений, ни внутреннее представление, ни размер байта. Поэтому мы считаем, что базовые типы занимают в памяти «единичные» области памяти, т.е. доступ к объекту может быть только по его адресу, как к единому целому, а доступ к отдельным байтам невозможен.

5. При работе с объектами важно знать их типы, поскольку они разделяются на объекты-переменные и объекты-функции, а первые, в свою очередь, необходимо разделять на модифицируемые и немодифицируемые. Информация о типах объектов определяется специальным отображением из имен в возможные типы:

$$\Gamma : \mathrm{IDs} \to \mathrm{Types}.$$

Как и все предыдущие компоненты, это отображение изменяется при работе программы. На основе Γ определяется система типов языка C-light, рассмотренная в п. 2.3. Отметим, что в данной версии семантики это отображение используется и для функций.

6. В языке C есть тип данных, в определении которого возможна явная рекурсия. Это структуры, соответствующие записям в языке Паскаль. Рекурсия возникает при определении динамических списков, когда один из членов структуры является указателем на этот же самый структурный тип. При исследовании свойств программ, в которых есть такие типы данных, простого отображения имен в типы может быть недостаточно. Поэтому используется специальное отображение Σ, сопоставляющее тегу структуры последовательность пар имен полей и типов полей структуры. Как и в [16], мы используем Σ_1 для обозначения последовательности первых элементов этих пар, т.е. все имена членов структуры. Последовательность типов членов обозначается как Σ_2.

7. В семантике языка C-light не предполагается какая-либо частичная компиляция. Следовательно, если для некоторого типа вводится синоним при помощи конструкции typedef, то все вхождения нового имени не заменяются вхождениями старого. Для определения соответствия между типами и их возможными синонимами используется компонента TD. Это отображение, сопоставляющее идентификатору имя уже существующего типа, связанное с ним typedef-декларацией.

8. Любое выражение языка C характеризуется, в первую очередь, своим типом и значением. Типы выражений определяются системой типов или статической семантикой, рассмотренной далее. При вычислении выражений необходимо запоминать значения (под)выражений. При возврате значений из функций на уровне операторов последнее, что происходит в семантике, — это обработка оператора return, переход в конец тела функции и выход из блока со всеми сопутствующими преобразованиями состояний. Возвращаемое значение также временно запоминается. Для этого используется компонента Val. Это специальная ячейка памяти, в которой может храниться значение любого типа, присутствующего в программе (кроме функций). В частности, если тип значения — void, то значение не определено. Более того, поскольку это значение реально вычисляется и его тип определяется системой типов (см. далее), то можно считать, что любое значение в этой ячейке — это пара: само значение и его тип. В отличие от семантики Норриша [16] это единственный случай, когда возникают такие пары. Помимо значений выражений здесь могут храниться несколько специальных значений абстрактной машины (п.2.4).

9. При определении первых шести компонент-семейств отмечалось, что индексация в них напрямую связана с уровнями вложенности в программе. Уровень вложенности текущей точки программы определяется специальной компонентой-флагом GLF. В начале работы программы этот флаг равен нулю. Затем при входе в любой блок он увеличивается на единицу, при выходе, соответственно, уменьшается.

Рассмотрим подробнее использование этого флага. Во-первых, инициализация переменных по умолчанию при отсутствии явных спецификаторов класса памяти определяется местом декларации, т. е. уровнем вложенности. Но главное использование заключается в разрешении областей определенности. Пусть программа выглядит так:

```
int x = 1, y = 2;        // GLF == 0
int main(void){          // GLF == 1
   double x = 3.0;
   int z;
   z = x+y*u;
   return 0;
}                        // GLF == 0
```

В данном примере два уровня вложенности. При обработке первой декларации все компоненты-семейства состоят из одного элемента, каждый из которых соответствует глобальному уровню. В теле функции `main` появляются вторые элементы, определяющие этот блок. Например IDs $= \{\{x, y\}^0, \{x, z\}^1\}$. Как будет обрабатываться оператор присваивания `z = x+y`? Детально семантика операции присваивания будет определена в п. 2.5, но уже сейчас можно понять следующее. Любой объект может быть использован только после точки его определения. Если в текущем блоке (на данном уровне вложенности) нет определения переменной, то скорее всего оно находится выше по тексту, в каком-либо охватывающем блоке (на более низком уровне вложенности). Если определения вообще нет, то по новому стандарту языка C — это ошибка. Таким образом, определить, что за объекты присутствуют в операторе присваивания, можно, двигаясь по компонентам изнутри наружу до первого совпадения. Для переменной `z` информация о ней будет найдена в элементах с индексами 1. Переменные с именем `x` присутствуют на обоих уровнях, но при поиске будет найдена внутренняя переменная вещественного типа. Переменная `y` будет найдена на внешнем уровне, поскольку в теле функции объектов с таким именем нет. Наконец, имя `u` не соответствует ни одному объекту в программе.

10. Кортеж BC, являющийся последней компонентой состояния абстрактной машины, — это информация о некотором блоке в программе (BlockContext). Первый элемент этого кортежа либо пуст, либо принадлежит IDs. Второй элемент либо пуст, либо принадлежит Adrs. Третий элемент либо пуст, либо принадлежит MeM. Четвертый элемент либо пуст, либо принадлежит MD. Пятый элемент либо пуст, либо принадлежит Г. Наконец последний элемент либо пуст, либо принадлежит Σ.

Таким образом, в этом кортеже может запоминаться часть информации о некотором состоянии. Это позволило очень просто задать обработку передачи управления по goto. Во многих известных работах избегают операторов передачи управления, поскольку их семантику сложно определить без знания контекста, в котором они находятся. Для любого другого оператора достаточно знать содержимое памяти машины и атрибуты объектов. Здесь же надо дополнительно знать, куда передается управление. Ранее предлагавшиеся подходы [5, 6] обладали недостатками. Например, приходилось запоминать тело программы/функции в некоторой компоненте либо задавать правила семантики для таких лексем, как фигурные скобки, по отдельности, либо существенно ограничивать передачу управления (с потерей многих, распространенных в программировании, способов ее использования). Новый подход свободен от всех этих недостатков и использует минимальное количество дополнительной информации. Заметим, что в большинстве конфигураций абстрактной машины этот кортеж вообще пуст, и лишь при выходе из блока, в котором находится нужная метка, эта компонента заполняется. Подробнее принцип ее работы рассмотрен в п. 2.7.

Вспомогательные функции. При работе абстрактной машины языка C-light могут использоваться специальные абстрактные функции. Они не являются функциями языка C-light и применяются только к состояниям машины. Перечислим их.

1) Labels — определение множества меток некоторого блока;
2) UnOpSem — реализация семантики одноместных операций;
3) BinOpSem — реализация семантики двухместных операций;
4) InstParms — реализация механизма подстановки фактических аргументов вместо формальных параметров при вызове функции;
5) ++ — инкремент флага GLF;
6) −− — декремент флага GLF.

У всех этих функций обязательным аргументом является текущее состояние машины, поэтому для удобства записи оно не изображается.

Функция Labels, помимо состояния, имеет в качестве аргумента тело некоторого блока, которое есть просто последовательность операторов. Результатом является множество имен меток, входящих в этот блок, кроме начинающихся с ключевых слов case и default. Знание этого множества в случае, когда управление дошло до конца тела блока, в сочетании со специальным значением, связанным с оператором goto (п. 2.7), позволит понять, куда передается управление — вперед или назад.

Язык C-light наследует у C все одноместные и двухместные операции, кроме побитных. Для того чтобы задать вычисление этих операций в выражениях единым образом, используются функции UnOpSem и BinOpSem [5, 6, 16]. У первой аргументами являются символ операции и значение (со своим типом), к которому эта операция применяется. У второй аргументами являются символ операции и два значения с их типами.

Напомним, что список деклараций параметров в определении функции — это фактически обычная последовательность объявлений объектов с локальной областью определенности. Но при вызове функции эти объекты инициализируются значениями фактических аргументов. Процесс обработки этой последовательности деклараций с одновременной инициализацией реализован в функции InstParms, аргументами которой являются список деклараций параметров функции и список значений.

Из предыдущих рассуждений понятно, что увеличение и уменьшение флага GLF происходит при входе в блок и выходе из блока, соответственно. Но помимо таких изменений флага необходимо также создавать и уничтожать элементы компонент-семейств с номером, равным текущему значению флага. Эти действия также возложены на функции $++$ и $--$. Заметим, что они не совсем симметричны. Функция $--$ удаляет элементы первых шести компонент с индексами, равными флагу в данном состоянии, а затем уменьшает флаг на единицу. Функция $++$ вначале увеличивает флаг на единицу, а затем копирует элементы из кортежа BC: i-й элемент кортежа добавляется с индексом, равным новому значению флага, к i-й компоненте. Затем кортеж BC очищается: каждый элемент становится пустым множеством. Напомним, что большую часть работы программы все элементы кортежа — пустые множества, поэтому функция $++$ будет просто создавать новые пустые элементы компонент.

Для обозначения множества всех состояний используем слово *States*. Для обозначения состояний используются греческие буквы σ, τ и другие с индексами. В работах [5, 6, 8] определялось понятие обновления состояния для простейшего случая, когда состояние — это одно отображение имен переменных в их значения. Теперь одновременное изменение нескольких компонент может записываться например так:

$$\sigma(\mathrm{ID}^i := \mathrm{ID}^i \cup \{\mathrm{new\_id}\}, \Gamma^i(\mathrm{new\_id} \leftarrow \mathsf{float}), \mathrm{Val} := 3).$$

Истинность в состоянии. В заключении этого раздела рассмотрим, как изменяется в новой семантике важное понятие истинности утверждения языка спецификаций в состоянии [5, 6]. Как и ранее, определение истинности задается индукцией по структуре утверждения, поэтому индуктивные переходы, соответствующие логическим связкам и кванторам, выглядят совершенно аналогично. Однако база индукции, в которой рассматриваются элементарные логические формулы, изменяется. Как обычно, истинность записывается в виде $\sigma \models p$:

- $\sigma \models v$ iff $\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(v)) \neq 0$, если $\Gamma_\sigma(v) = \mathsf{int}$;
- $\sigma \models v$ iff $\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(v)) = \mathsf{TRUE}$, если $\Gamma_\sigma(v) = \mathsf{bool}$;
- $\sigma \models P(e_1, ..., e_n)$ iff $P(v_1, ..., v_n) = \mathsf{TRUE}$, если P — это предикатный символ, а v_i — результат вычисления выражения e_i в состоянии σ (семантика вычисления выражений описана в п. 2.5).

Если $\sigma \models p$, то говорим, что p *выполняется* в σ, а также используем понятие смысла утверждения, определенного как

$$\|p\| = \{\sigma | \sigma - \text{состояние и } \sigma \models p\}.$$

Мы говорим, что утверждение p *истинно* или *выполняется*, если $\|p\| = States$. Лемма 2 из [6] выполняется и в новой семантике.

2.3. Система типов

Рассмотрим подробнее синтаксис выражений языка C-light, одновременно определяя правила типизации. Тем самым мы определим статическую семантику языка C-light, или *систему типов*. В статической семантике не рассматриваются операторы, поскольку, хотя у них и есть значения — преобразователи состояний, эти значения только неявно присутствуют в абстрактной машине.

Следует отметить, что в нашем случае, в отличие от работы [16], термин *система типов* более уместен, нежели термин *статическая*

семантика. Там система типов определялась достаточно независимо от динамической семантики, определяющей поведение программы. В частности, для определения типов идентификаторов переменных и функций использовались специальные контексты Γ, Φ и Σ, которые являлись отображениями идентификаторов переменных, функций и структур, соответственно, в их типы. А затем эти три контекста отождествлялись с соответствующими компонентами абстрактной машины и утверждалось, что на вход динамической семантике подается программа, в которой все значения сопровождаются их типами. Сами типы заранее определяются статической семантикой, т.е. в этой работе неявно присутствует процесс некоторой частичной компиляции, и статическая семантика выполняет функцию отсева программ, некорректных с точки зрения типов. Тем самым статическая семантика должна сопровождаться правилами обработки деклараций, но они есть только в динамической семантике, которая использует результат работы статической. Мы избегаем такого подхода, и на вход динамической семантике подается не промежуточный код абстрактного компилятора, а исходный код на языке C-light. Тип выражения определяется непосредственно в момент его вычисления в контексте текущего состояния вычислительной машины. Поэтому в нашей системе типов Γ — это изначально не абстрактное сопоставление переменным их типов, а компонента состояния, формируемая на основе деклараций программы. Как станет ясно далее, с точки зрения динамической семантики входной программой может быть любой оператор и даже просто выражение. Естественно, смысл такой «программы» существенно зависит от начального состояния. Если в нем вообще нет информации об объектах, на которые есть ссылки в этом фрагменте, то это ошибка. А если информация в начальном состоянии есть, она может быть любой, в том числе и неправильной. Эта особенность, отличающая данную работу от [16], подробнее обсуждается при доказательстве основной теоремы о корректности.

Как и в C, значением выражения может быть ссылка на объект в памяти — так называемое L-значение (lvalue). В правилах динамической семантики для деклараций в качестве типа переменой будет сразу записываться не T, а $\mathrm{lv}[T]$.

Выражения языка C-light определяются по индукции. Следовательно, систему типов можно представить как формальную систему вывода. Для изображения чисел используется символ n, для символов — c, для переменных, функций и членов структур — символ id. Выражение отде-

ляется от его типа двоеточием. Базой являются аксиомы для констант и переменных, а также для операции sizeof.

Первая группа из семи аксиом стандартным образом определяет типы числовых констант:

$$\begin{array}{ll} n : \mathsf{signed\ int} & n_1.n_2[\mathtt{E}\, n_3] : \mathsf{double} \\ n\mathtt{L} : \mathsf{long} & n_1.n_2[\mathtt{E}\, n_3]\mathtt{F} : \mathsf{float} \\ n\mathtt{U} : \mathsf{unsigned\ int} & n_1.n_2[\mathtt{E}\, n_3]\mathtt{L} : \mathsf{long\ double} \\ n\mathtt{S} : \mathsf{short\ int} & \end{array}$$

Вторая группа из четырех аксиом задает типы символьных и строковых констант (литералов):

$$\begin{array}{ll} \mathtt{'}c\mathtt{'} : \mathsf{char} & \mathtt{"}c_1 \ldots c_n\mathtt{"} : \mathsf{char}[n+1] \\ \mathtt{L'}c_1c_2\mathtt{'} : \mathsf{wchar\_t} & \mathtt{L"}c_1 \ldots c_n\mathtt{"} : \mathsf{wchar\_t}[n/2+1] \end{array}$$

Две аксиомы используются для нулевого указателя, который имеет специальное назначение. Т.е. ноль — это полиморфная константа, которая может быть не только целым числом, но и автоматически приводится к любому ссылочному типу:

$$\mathsf{NULL} : \mathsf{void} * \qquad 0 : T*$$

Тип объекта, именуемого идентификатором id, определяется компонентой Γ, и в отличие от [16] не вводится такая же компонента для имен функций:

$$\mathsf{id} : \Gamma(\mathsf{id})$$

Последние две аксиомы являются следствием нового стандарта языка C, поскольку с введением массивов переменной длины операция sizeof уже не является константой, вычисляемой на этапе компиляции. По определению операция возвращает значения специального типа size_t, но этот тип всегда просто синоним типа, достаточного для хранения размеров любых возможных типов (в конкретной архитектуре). Поэтому мы считаем, что это беззнаковый целый:

$$\mathtt{sizeof}(expn) : \mathsf{unsigned\ int} \qquad \mathtt{sizeof}(type\_spec) : \mathsf{unsigned\ int}$$

Далее все типы выражений определяются правилами вывода. Первые два относятся к операциям адресации и косвенной адресации. Отметим сразу, что нет отдельного правила для индексных выражений,

так как выражение `a[i]` есть просто удобная запись для $*(\mathtt{a}+\mathtt{i})$.

$$\frac{e : \mathrm{lv}[T]}{\&e : T*} \qquad \frac{e : T* \qquad T \neq \mathsf{void}}{*e : \mathrm{lv}[T]}$$

Очевидна связь между L-значениями и просто значениями, которая нарушается только для массивов, являющихся немодифицируемыми L-значениями. Во всех выражениях, где требуется значение, идентификатор массива понимается как указатель на первый элемент, кроме случая операций sizeof и &.

$$\frac{e : \mathrm{lv}[T] \qquad T - \text{не массив}}{e : T} \qquad \frac{e : \mathrm{lv}[T[n]]}{e : T*}$$

Структура может быть либо L-значением, либо просто значением, например, при возврате из функции. Возврат произвольной структуры в качестве значения функции может привести к нежелательной ситуации: если полем такой структуры является массив, то в результате применения операции выбора элемента (точка) к этому значению можно получить массив, который является просто значением. Как видно из второго правила, для структур такая ситуация запрещена. Отметим сразу, что запись вида `ptr->f` обрабатывается как `(*ptr).f`.

$$\frac{e : \mathrm{lv}[\mathsf{struct}\ s] \qquad (\mathtt{id}, T) \in \Sigma(s)}{e.\mathtt{id} : \mathrm{lv}[T]}$$

$$\frac{e : \mathsf{struct}\ s \qquad (\mathtt{id}, T) \in \Sigma(s) \qquad T - \text{не массив}}{e.\mathtt{id} : T}$$

Процесс приведения типов может быть опасным, если, например, происходит потеря точности или значимости. Поэтому накладывается ограничение на приведение типов для указателей, а именно: один из типов в приведении обязательно должен быть void. Для приведения типов используется стандартный синтаксис языка C.

$$\frac{e : T_0 \qquad (T_0 \text{ и } T - \text{скалярные типы}) \vee (T = \mathsf{void})}{(T)e : T}$$

Для корректной обработки вызовов функций и присваиваний необходимо ввести отношение приводимости по умолчанию (**implicit coercibility**). В противном случае в правилах для операции присваивания и вызова

функции пришлось бы явно записывать требования точного согласования типов аргументов. Это отношение, обозначаемое как IC, будет отношением эквивалентности [16]. Во втором правиле одним из типов может быть тип bool.

$$\vdash IC(\mathsf{void}*, T*) \qquad \frac{T_1 \text{ и } T_2 - \text{арифметические типы}}{IC(T_1, T_2)}.$$

Тогда правило для вызова функции выглядит как

$$\frac{e : T_1 \times ... \times T_n \to T \qquad \forall i.1 \le i \le n \Rightarrow \exists T'.(e_i : T' \wedge IC(T_i, T'))}{e(e_1, ..., e_n) : T}.$$

Что касается операций языка C-light, то правила для них могут быть достаточно громоздкими, потому что над аргументами происходят стандартные преобразования типов. Поэтому информация об окончательном типе выражения определяется специальными предикатами. Они определяются в соответствии со стандартом, поэтому их реализация не представляет здесь особого интереса. Для обозначения унарных операций используем символ $\square$, для бинарных операций — символ $\odot$. Заметим, что операции постфиксного инкремента/декремента рассмотрены отдельно:

$$\frac{e : T}{\square e : T} \qquad \frac{e_1 : T_1 \qquad e_2 : T_2 \qquad P_\odot(T_1, T_2, T)}{e_1 \odot e_2 : T},$$

$$\frac{e_0 : T_0 - \text{скалярный} \qquad e_1 : T_1 \qquad e_2 : T_2 \qquad P_?(T_1, T_2, T)}{e_0 ? e_1 : e_2 : T}.$$

В операциях присваивания необходимо, чтобы выражение в правой части было модифицируемым L-значением. Правила для простого и составного присваиваний выглядят как

$$\frac{e_1 : \mathrm{lv}[T] \qquad e_2 : T_0 \qquad IC(T_0, T) \qquad T - \text{не массив}}{e_1 = e_2 : T},$$

$$\frac{e_1 : \mathrm{lv}[T] \qquad e_1 \odot e_2 : T_0 \qquad IC(T_0, T) \qquad T - \text{не массив}}{e_1 \odot = e_2 : T}.$$

Операции постфиксных инкремента и декремента определяются одинаково. Заметим, что префиксный инкремент и декремент можно обрабатывать с помощью правила для составного присваивания, так как выражения `++i` и `--i` есть эквивалентные записи для `i+=1` и `i-=1`.

$$\frac{e : \mathrm{lv}[T] \qquad T - \text{скалярный}}{e{+}{+} : T}.$$

Логические выражения строятся из выражений типа bool с помощью логических связок и кванторов, как в логике первого порядка. Далее все выражения типа bool будем называть утверждениями.

2.4. Конфигурации абстрактной машины языка C-light

Операционная семантика языка программирования представляется в виде множеств пар конфигураций абстрактной вычислительной машины, связанных отношениями перехода. Как и состояния, конфигурации могут быть различными в зависимости от языка программирования и целей. В нашем случае используется классическое понятие, поскольку все особенности языка и вычислений «инкапсулированы» в состояниях.

Определение. Конфигурация абстрактной вычислительной машины языка C-light — это пара $\langle P, \sigma \rangle$: программа P и состояние σ. Пустая программа обозначается символом E, и записи $E\ A$ и $A\ E$ являются эквивалентными обозначениями для программы A.

Семантика программы определяется индукцией по структуре программы в терминах отношений переходов. Для языка C-light предлагаются два бинарных отношения перехода, а именно:

1) отношение для вычисления выражений: $\rightarrow_e$,

2) отношение для обработки операторов: $\rightarrow_s$.

Отметим, что в отличие от работ [5, 16] отсутствует отдельное отношение $\rightarrow_v$ для деклараций, поскольку этих двух отношений достаточно для обработки любого объявления.

Итак, опишем все аксиомы и правила динамической операционной семантики языка C-light. Аксиомы будут изображаться в виде пар конфигураций, связанных одним из отношений перехода, и запись $\langle A, \sigma \rangle \rightarrow \langle B, \tau \rangle$ означает, что один шаг исполнения фрагмента A исходной программы, начинающийся в состоянии σ, приводит в состояние τ, причем B — тот фрагмент исходной программы, который остается для исполнения. Заметим, что если бы в исходной программе отсутствовали переходы "goto назад", то фрагмент B был бы частью A. Посылки в правилах семантики отделяются от заключений чертой.

Далее для удобства значение флага вложенности, как индекса составных частей, будет опускаться, если речь идет о части с текущим номером. Символ текущего состояния часто также легко определить из конфигурации. Например, вместо $\Gamma_\sigma^{\mathrm{GLF}}(\mathrm{id}) = \mathsf{int}$ может быть запись $\Gamma(\mathrm{id}) = \mathsf{int}$. Поиск в компонентах происходит изнутри наружу (т.е. по

убыванию индексов), поэтому если номер блока не важен, такая запись не приводит к двусмысленности.

Пустой фрагмент будем обозначать символом ϵ. Пустым фрагментом может быть как пустая программа, так и пустое выражение. Неопределенное значение будем обозначать символом ω.

Наконец перечислим специальные значения абстрактной машины, которые могут храниться в компоненте Val, наряду со значениями программных типов. В отличие от обычных, эти значения не имеют типов и, как правило, сигнализируют о том или ином событии в ходе работы программы, т.е. их использование отчасти похоже на механизм обработки исключений в некоторых языках программирования (например C++).

1) OkVal — нормальное завершение работы оператора;
2) GoVal(L) — был встречен оператор `goto L`;
3) BreakVal — был встречен оператор `break`;
4) ContVal — был встречен оператор `continue`;
5) RetVal(e) — был встречен оператор `return e`;
6) CaseVal(e) — был встречен оператор `switch(e){...}`;
7) ω — неопределенность на уровне вычисления выражений;
8) Fail — протягивание значения ω на уровень вычисления операторов, что соответствует аварийному останову.

Подробнее эти значения будут объясняться по мере описания семантики операторов, в которых они могут возникать.

2.5. Семантика выражений

Небольшое усложнение правил семантики, а также избавление от недетерминизма позволило не вводить контекст вычисления выражений, использовавшийся в [5, 16].

Основным внешним правилом становится правило протягивания неопределенного значения:

$$\frac{\mathrm{Val}_\sigma = \omega}{\langle e, \sigma\rangle \to_e \langle \epsilon, \sigma\rangle}. \tag{1}$$

В [5, 16] для большинства выражений и операторов использовались по два правила. В одном рассматривалась нормальная работа соответствующей конструкции, в другом моделировалось неопределенное поведение, которое означает тот или иной тип ошибки периода исполнения

программы. Для краткости в данном описании правила для неопределенности в выражениях не рассматриваются. Эти правила практически не изменяются по сравнению с [5, 16], при возникновении ошибки при вычислении в компоненту Val заносится значение ω, и далее все последующие вычисления игнорируются в соответствии с (1).

Значения из памяти. Константы:

$$\frac{\sigma \models const : \tau}{\langle const, \sigma \rangle \to_e \langle \epsilon, \sigma(\mathrm{Val} := (const, \tau)) \rangle}. \tag{2}$$

Переменные и L-значения:

$$\frac{\Gamma_\sigma(\mathsf{id}) = \mathrm{lv}[\tau] \qquad \tau - \text{не массив и не функция}}{\langle \mathsf{id}, \sigma \rangle \to_e \langle \epsilon, \sigma(\mathrm{Val} := (\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(\mathsf{id})), \tau)) \rangle}. \tag{3}$$

Массивы:

$$\frac{\Gamma_\sigma(\mathsf{arr\_id}) = \mathrm{lv}[\tau[n]]}{\langle \mathsf{arr\_id}, \sigma \rangle \to_e \langle \epsilon, \sigma(\mathrm{Val} := (\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(\mathsf{arr\_id})), \tau*)) \rangle}. \tag{4}$$

Индексное выражение:

$$\frac{\begin{array}{c}\sigma \models a : \mathrm{lv}[\tau[n]] \qquad \tau - \text{не массив} \\ \langle e, \sigma \rangle \to_e^* \langle \epsilon, \sigma_1 \rangle \qquad \mathrm{Val}_{\sigma_1} = (v, \tau') \qquad \tau' - \text{скалярный} \\ \langle a, \sigma_1 \rangle \to_e^* \langle \epsilon, \sigma_2 \rangle \qquad \mathrm{Val}_{\sigma_2} = (c, \tau*)\end{array}}{\langle a\,[e]\,, \sigma \rangle \to_e \langle \epsilon, \sigma_2(\mathrm{Val} := (\mathrm{MD}_{\sigma_2}(c + v), \tau)) \rangle}. \tag{5}$$

Выражение выбора элемента:

$$\frac{\Gamma_\sigma(s) = \mathrm{lv}[\mathsf{struct}(\tau_1, ..., \tau_n)] \qquad (\mathsf{id}, \tau) \in \Sigma_\sigma(s)}{\langle s.\mathsf{id}, \sigma \rangle \to_e \langle \epsilon, \sigma(\mathrm{Val} := (\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(s)).\mathsf{id}, \tau)) \rangle}. \tag{6}$$

Выражение косвенной адресации:

$$\frac{\begin{array}{c}\sigma \models e : \mathrm{lv}[\tau*] \qquad \tau - \text{не массив и не функция} \\ \langle e, \sigma \rangle \to_e^* \langle \epsilon, \sigma_1 \rangle \qquad \mathrm{Val}_{\sigma_1} = (v, \tau*)\end{array}}{\langle *e, \sigma \rangle \to_e \langle \epsilon, \sigma_1(\mathrm{Val} := (\mathrm{MD}_{\sigma_1}(v), \tau)) \rangle}. \tag{7}$$

Выражение адресации:

$$\frac{\sigma \models e : \mathrm{lv}[\tau] \qquad \tau - \text{не функция} \qquad \langle e, \sigma \rangle \to_e^* \langle \epsilon, \sigma' \rangle}{\langle \&e, \sigma \rangle \to_e \langle \epsilon, \sigma'(\mathrm{Val} := (\mathrm{MeM}_\sigma(e), \tau*)) \rangle}. \tag{8}$$

Операции, выдающие значения. Ранее были введены специальные функции UnOpSem и BinOpSem, реализующие семантики одноместных и двухместных операций соответственно. Пусть StdBin обозначает множество всех двухместных операций языка C-light, а BinSeq обозначает трехэлементное множество, состоящее из операции логического И, операции логического ИЛИ и операции последовательного вычисления (запятая).

$$\frac{\begin{array}{c}\odot \in \mathrm{StdBin} \setminus \mathrm{BinSeq} \\ \langle e_2, \sigma_0\rangle \to_e^* \langle \epsilon, \sigma_1\rangle \qquad \langle e_1, \sigma_1\rangle \to_e^* \langle \epsilon, \sigma_2\rangle \\ \mathrm{Val}_{\sigma_1} = (v_2, \tau_2) \qquad \mathrm{Val}_{\sigma_2} = (v_1, \tau_1) \qquad IC(\tau_1, \tau_2)\end{array}}{\langle (e_1 \odot e_2), \sigma\rangle \to_e \langle \epsilon, \sigma_2(\mathrm{Val} := \mathrm{BinOpSem}(\odot, v_1, \tau_1, v_2, \tau_2))\rangle}. \tag{9}$$

$$\frac{\square \in \{+, -, !, (.), \mathsf{sizeof}\} \qquad \langle e, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle}{\langle \square e, \sigma\rangle \to_e \langle \epsilon, \sigma'(\mathrm{Val} := \mathrm{UnOpSem}(\square, \mathrm{Val}_{\sigma'}))\rangle}. \tag{10}$$

$$\frac{\langle e, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \qquad \mathrm{Val}_{\sigma'} = (v_0, \tau_0) \qquad \gamma_{\tau_0,\tau}(v_0) = v}{\langle (\tau) e, \sigma\rangle \to_e \langle \epsilon, \sigma'(\mathrm{Val} := (v, \tau))\rangle}. \tag{11}$$

Напомним, что операции приведения типа моделируются семействами функций γ_{τ_1,τ_2}, предложенными в [5]. Также отметим, что не все одноместные операции вычисляются посредством общей функции в (10).

К операциям, выдающим значения, относится и операция последовательного вычисления:

$$\frac{\begin{array}{c}\mathit{head} \text{ не содержит не верхнем уровне операцию “,”} \\ \langle \mathit{head}, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle\end{array}}{\langle \mathit{head}, \mathit{tail}, \sigma\rangle \to_e \langle \mathit{tail}, \sigma'(\mathrm{Val} := \mathrm{OkVal})\rangle}. \tag{12}$$

Заметим, что занесение значения OkVal в состояние σ' в заключении правила соответствует тому, как в определении этой операции значение левого, уже вычисленного, выражения приводится к типу void, т.е. просто игнорируется.

Логические операции И и ИЛИ, а также условная операция выделены в отдельную группу. Как известно, выражения, содержащие эти операции, могут вычисляться не полностью, поэтому такие операции являются контрольными точками. Для вычисления этих операций вводится промежуточная форма **OrAnd**.

$$\frac{\langle e_1, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \qquad \mathrm{Val}_{\sigma'} = (0, \tau) \qquad \tau - \text{скалярный тип}}{\langle e_1 \&\& e_2, \sigma\rangle \to_e \langle \epsilon, \sigma'(\mathrm{Val} = (\mathsf{FALSE}, \mathsf{bool}))\rangle}. \tag{13}$$

$$\frac{\langle e_1,\sigma\rangle \to_e^* \langle \epsilon,\sigma'\rangle \quad \mathrm{Val}_{\sigma'}=(v,\tau) \quad \tau - \text{скалярный тип} \quad v\neq 0}{\langle e_1 \,||\, e_2,\sigma\rangle \to_e \langle \epsilon,\sigma'(\mathrm{Val}=(\mathsf{TRUE},\mathsf{bool}))\rangle}. \tag{14}$$

$$\frac{\langle e_1,\sigma\rangle \to_e^* \langle \epsilon,\sigma'\rangle \quad \mathrm{Val}_{\sigma'}=(v,\tau) \quad \tau - \text{скалярный тип} \quad v\neq 0}{\langle e_1 \,\&\&\, e_2,\sigma\rangle \to_e \langle \mathbf{OrAnd}(e_2),\sigma'\rangle}. \tag{15}$$

$$\frac{\langle e_1,\sigma\rangle \to_e^* \langle \epsilon,\sigma'\rangle \qquad \mathrm{Val}_{\sigma'}=(0,\tau) \qquad \tau - \text{скалярный тип}}{\langle e_1 \,||\, e_2,\sigma\rangle \to_e \langle \mathbf{OrAnd}(e_2),\sigma'\rangle}. \tag{16}$$

$$\frac{\langle e,\sigma\rangle \to_e^* \langle \epsilon,\sigma'\rangle \qquad \mathrm{Val}_{\sigma'}=(0,\tau) \qquad \tau - \text{скалярный тип}}{\langle \mathbf{OrAnd}(e),\sigma\rangle \to_e \langle \epsilon,\sigma'(\mathrm{Val}=(\mathsf{FALSE},\mathsf{bool}))\rangle}. \tag{17}$$

$$\frac{\langle e,\sigma\rangle \to_e^* \langle \epsilon,\sigma'\rangle \quad \mathrm{Val}_{\sigma'}=(v,\tau) \quad \tau - \text{скалярный тип} \quad v\neq 0}{\langle \mathbf{OrAnd}(e),\sigma\rangle \to_e \langle \epsilon,\sigma'(\mathrm{Val}=(\mathsf{TRUE},\mathsf{bool}))\rangle}. \tag{18}$$

$$\frac{\begin{array}{c}\sigma \models (e_1:\tau_1)\wedge(e_2:\tau_2)\wedge P_?(\tau_1,\tau_2,\tau) \qquad \langle e_0,\sigma\rangle \to_e^* \langle \epsilon,\sigma'\rangle \\ \mathrm{Val}_{\sigma'}=(v,\tau_0) \qquad \tau_0 - \text{скалярный тип} \qquad v\neq 0\end{array}}{\langle e_0 ? e_1 : e_2,\sigma\rangle \to_e \langle (\tau)\, e_1,\sigma'\rangle}. \tag{19}$$

$$\frac{\begin{array}{c}\sigma \models (e_1:\tau_1)\wedge(e_2:\tau_2)\wedge P_?(\tau_1,\tau_2,\tau) \qquad \langle e_0,\sigma\rangle \to_e^* \langle \epsilon,\sigma'\rangle \\ \mathrm{Val}_{\sigma'}=(0,\tau_0) \qquad \tau_0 - \text{скалярный тип}\end{array}}{\langle e_0 ? e_1 : e_2,\sigma\rangle \to_e \langle (\tau)\, e_2,\sigma'\rangle}. \tag{20}$$

Операции с побочными эффектами. Как и в C, любое изменение содержимого памяти в программе на языке C-light происходит посредством побочных эффектов. Выражение характеризуется в первую очередь своим значением и типом. Поэтому даже простая операция присваивания вызывает побочный эффект[2].

Все побочные эффекты, генерируемые при вычислении выражений, вычислятся на месте в отличие от [5, 16].

$$\frac{\begin{array}{c}\sigma \models e_1 : \mathrm{lv}[\tau_1] \qquad \tau_1 - \text{ не массив} \\ \langle e_2,\sigma_0\rangle \to_e^* \langle \epsilon,\sigma_1\rangle \qquad \langle e_1,\sigma_1\rangle \to_e^* \langle \epsilon,\sigma_2\rangle \\ \mathrm{Val}_{\sigma_1}=(v_2,\tau_2) \quad \mathrm{Val}_{\sigma_2}=(v_1,\tau_1) \quad \mathrm{MeM}_{\sigma_2}(e_1)=c \quad IC(\tau_1,\tau_2)\end{array}}{\langle e_1 \odot{=} e_2,\sigma_0\rangle \to_e \langle \epsilon,\sigma_2''\rangle}, \tag{21}$$

где $\sigma_2'' = \sigma_2(\mathrm{MD}(c \leftarrow v_{res}))(\mathrm{Val} := (v_{res},\tau_1))$,
$v_{res} = \mathrm{BinOpSem}(\odot, v_1, \tau_1, v_2, \tau_2)$.

[2]Изменения памяти при декларациях рассмотрены в п. 2.6.

Таким образом, возможные в e_1 побочные эффекты не дублируются.

$$\frac{\begin{array}{c}\sigma \models e_1 : \mathrm{lv}[\tau_1] \qquad \tau_1 - \text{ не массив} \\ \langle e_2, \sigma_0\rangle \to_e^* \langle \epsilon, \sigma_1\rangle \qquad \langle e_1, \sigma_1\rangle \to_e^* \langle \epsilon, \sigma_2\rangle \\ \mathrm{Val}_{\sigma_1} = (v_2, \tau_2) \qquad \mathrm{MeM}_{\sigma_2}(e_1) = c \qquad IC(\tau_1, \tau_2)\end{array}}{\langle e_1 = e_2, \sigma_0\rangle \to_e \langle \epsilon, \sigma_2''\rangle}, \tag{22}$$

где $\sigma_2'' = \sigma_2(\mathrm{MD}(c \leftarrow v_2))(\mathrm{Val} := (\gamma_{\tau_2,\tau_1}(v_2), \tau_1))$.

$$\frac{\sigma \models e : \mathrm{lv}[\tau] \qquad \tau - \text{ не массив}}{\langle {+}{+}e, \sigma\rangle \to_e \langle e \mathrel{+}= 1, \sigma\rangle}, \tag{23}$$

т.е. префиксный инкремент/декремент переписывается на месте в эквивалентный фрагмент.

$$\frac{\begin{array}{c}\sigma \models e : \mathrm{lv}[\tau] \qquad \tau - \text{ не массив} \\ \langle e, \sigma\rangle \to_e^* \langle \epsilon, \sigma_1\rangle \qquad \mathrm{Val}_{\sigma_1} = (v, \tau) \qquad \mathrm{MeM}_{\sigma_1}(e) = c\end{array}}{\langle e{+}{+}, \sigma\rangle \to_e \langle \epsilon, \sigma_1(\mathrm{MD}(c \leftarrow (v+1)))\rangle}. \tag{24}$$

Очевидно, что без использования адресов задать семантику операций присваивания языка C-light было бы невозможно, поскольку левая часть в общем случае может быть выражением с побочными эффектами.

Операции над динамической памятью. Хотя операция выделения памяти относится к операциям, выдающим значения, рассмотрим ее вместе с операцией освобождения памяти.

$$\frac{\tau - \text{ не функция}}{\langle \texttt{new } \tau, \sigma\rangle \to_e \langle \epsilon, \sigma'\rangle}, \tag{25}$$

где $\sigma' = \sigma(\mathrm{Adr}^0 \leftarrow \mathrm{Adr}^0 \cup \{nc\})(\mathrm{MD}^0(nc \leftarrow 0))(\mathrm{Val} := (nc, \tau*))$,
nc – новая константа.

$$\frac{\begin{array}{c}\tau - \text{ не функция} \\ \langle size, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \qquad \mathrm{Val}_{\sigma'} = (v, \mathsf{unsigned\ int}) \qquad v \geq 0\end{array}}{\langle \texttt{new } \tau\texttt{[}size\texttt{]}, \sigma\rangle \to_e \langle \epsilon, \sigma''\rangle}, \tag{26}$$

где $\sigma'' = \sigma(\mathrm{Adr}^0 \leftarrow \mathrm{Adr}^0 \cup \{nc+0\})(\mathrm{MD}^0(nc+0 \leftarrow 0))$
$\ldots$
$(\mathrm{Adr}^0 \leftarrow \mathrm{Adr}^0 \cup \{nc+v-1\})(\mathrm{MD}^0(nc+v-1 \leftarrow 0))$
$(\mathrm{Val} := (nc+0, \tau*))$,
nc – новая константа.

Таким образом, выделение памяти означает появление нового адреса для простого объекта и набора новых адресов для массива.

$$\frac{\sigma \models e : \mathrm{lv}[\tau*] \qquad \tau - \text{ не функция}}{\langle \texttt{delete } e, \sigma\rangle \to_e \langle \epsilon, \sigma'\rangle}, \tag{27}$$

где $\sigma' = \sigma(\mathrm{Adr}^0 \leftarrow \mathrm{Adr}^0 \setminus \{\mathrm{MD}^i(\mathrm{MeM}^i(e))\})$, а $i = \mathrm{GLF}_\sigma$.

$$\frac{\sigma \models e : \mathrm{lv}[\tau*] \qquad \tau - \text{ не функция}}{\langle \texttt{delete[] } e, \sigma\rangle \to_e \langle \epsilon, \sigma\rangle}, \tag{28}$$

где $\sigma' = \sigma(\mathrm{Adr}^0 \leftarrow \mathrm{Adr}^0 \setminus \{\mathrm{MD}^i(\mathrm{MeM}^i(e))\})$, а $i = \mathrm{GLF}_\sigma$.

Освобождение памяти в обоих случаях моделируется простым удалением первого адреса. Удалить все адреса для массива невозможно, поскольку в операции delete размер удаляемого массива не указывается, но это не влияет на корректность, ведь адреса — это неинтерпретируемые константы. Более того мы полагаем по определению, что всегда выделяется новая память, а свободная память всегда есть в достаточном количестве. Обратим внимание на два нюанса: 1) при освобождении нет необходимости изменять компоненту MD, поскольку удаление адреса автоматически изменяет область определенности MD; 2) при освобождении не изменяется значение самого указателя, что соответствует семантике операции delete в общем случае.

Вызовы функций. При вызове любой функции происходит переход к ее телу с передачей аргументов. В C-light, как и в C, аргументы передаются только по значению. Для упрощения семантика вызова функции представлена двумя правилами: в первом происходит вычисление значений аргументов, во втором — непосредственно переход с передачей полученных значений.

$$\frac{\begin{array}{c}\sigma \models f : \tau_1 \times ... \times \tau_n \to \tau_r \\ \langle e_n, \sigma_0\rangle \to_e^* \langle \epsilon, \sigma_1\rangle \; ... \; \langle e_i, \sigma_i\rangle \to_e^* \langle \epsilon, \sigma_{i+1}\rangle \; ... \; \langle e_1, \sigma_n\rangle \to_e^* \langle \epsilon, \sigma_{n+1}\rangle \\ \mathrm{Val}_{\sigma_1} = (v_n, \tau'_n) \quad ... \quad \mathrm{Val}_{\sigma_{i+1}} = (v_i, \tau'_i) \quad ... \quad \mathrm{Val}_{\sigma_{n+1}} = (v_1, \tau'_1) \\ IC(\tau_n, \tau'_n) \quad ... \quad IC(\tau_i, \tau'_i) \quad ... \quad IC(\tau_1, \tau'_1)\end{array}}{\langle f(e_1, ..., e_i, ..., e_n), \sigma_0\rangle \to_e \langle \mathbf{FCall}(f)(v_1, ..., v_i, ..., v_n), \sigma_{n+1}\rangle}. \tag{29}$$

$$\frac{\begin{array}{c}\sigma \models f : \tau_1 \times ... \times \tau_n \to \tau_r \\ \tau\ f(parms)\{S\} - \text{определение функции } f \\ \langle S, \text{InstParms}(\sigma(\text{GLF}\ {+}{+}), \text{parms}, [v_1 ... v_n])\rangle \to_s^* \langle \epsilon, \sigma' \rangle \\ \text{Val}_{\sigma'} = \text{RetVal}(v, \tau') \vee \text{OkVal} \qquad IC(\tau_r, \tau')\end{array}}{\langle \mathbf{FCall}(f)(v_1, ..., v_n), \sigma\rangle \to_e \langle \epsilon, \sigma'(\text{Val} := (v, \tau_r))(\text{GLF}\ {-}{-})\rangle}. \tag{30}$$

Для того чтобы последнее правило корректно обрабатывало и вызовы функций, не возвращающих значения (т.е. процедур), достаточно потребовать совместимости значения OkVal с типом void.

2.6. Семантика деклараций

Рассмотрим аксиомы и правила для деклараций аннотаций, типов и объектов. Инициализатором объекта может быть выражение, поэтому могут срабатывать правила для отношения $\to_e$.

Аннотации и типы.

$$\langle \texttt{/\%}\ annotation\ \texttt{\%/}, \sigma\rangle \to_s \langle \epsilon, \sigma\rangle. \tag{31}$$

$$\langle \texttt{typedef}\ type\_spec\ \texttt{Id;}, \sigma\rangle \to_s \langle \epsilon, \sigma(\text{TD}(\texttt{Id} \leftarrow \text{TD}(type\_spec)))\rangle. \tag{32}$$

$$\langle \texttt{typedef struct \{}fields\texttt{\} Id;}, \sigma\rangle \to_s \langle S, \sigma\rangle, \tag{33}$$

где $S \equiv$ `struct new_tag {`*fields*`}; typedef struct new_tag Id;`
а `new_tag` — новый уникальный идентификатор. Новые теги для безымянных структурных типов вводятся для того, чтобы отображение Σ было определено.

Уточнение класса памяти и декомпозиция.

$$\langle type\_spec\ drator\texttt{;}, \sigma\rangle \to_s \langle storage\ type\_spec\ drator\texttt{;}, \sigma\rangle, \tag{34}$$

где $storage = \begin{cases}\mathsf{static}, & \text{если } \text{GLF}_\sigma = 0, \\ \mathsf{auto}, & \text{иначе.}\end{cases}$

$$\langle \texttt{register}\ type\_spec\ drator\texttt{;}, \sigma\rangle \to_s \langle \texttt{auto}\ type\_spec\ drator\texttt{;}, \sigma\rangle. \tag{35}$$

$$\langle storage_{opt}\ type\_spec\ drator\texttt{,}\ drator\_list\texttt{;}, \sigma\rangle \to_s \langle S, \sigma\rangle, \tag{36}$$

где $S \equiv storage_{opt}\ type\_spec\ drator\texttt{;}\ storage_{opt}\ type\_spec\ drator\_list\texttt{;}$ Т.е. расставляются спецификаторы класса памяти по умолчанию, а цепочки деклараторов разбиваются на отдельные декларации.

Простые переменные. В языке C-light спецификацией типа можно определить только переменную базового типа, структуру или перечисление. Остальные типы определяются модификатором. Поэтому в посылках правил для простых переменных есть специальная посылка относительно типа.

$$\frac{\mathrm{TD}(\tau) - \text{не структура и не перечисление}}{\langle storage\ \tau\ \texttt{id;}, \sigma\rangle \to_s \langle \epsilon, \sigma'\rangle}, \tag{37}$$

$$\begin{aligned}
\text{где}\ \ \sigma' \ &= \ \sigma(\mathrm{ID}^i \leftarrow \mathrm{ID}^i \cup \{\texttt{id}\})(\mathrm{Adr}^i \leftarrow \mathrm{Adr}^i \cup \{nc\})(\mathrm{MeM}(\texttt{id} \leftarrow nc)) \\
&\quad (\Gamma^i(\texttt{id} \leftarrow \tau))(\mathcal{I}nit(\texttt{id}, storage))(\mathrm{Val} := \mathrm{OkVal}), \\
\text{а}\quad i \ &= \ \begin{cases} 0, & \text{если } storage \equiv \mathsf{static}, \\ \mathrm{GLF}_\sigma, & \text{если } storage \equiv \mathsf{auto}. \end{cases}
\end{aligned}$$

$$\frac{\begin{array}{c}\mathrm{TD}(\tau) - \text{не структура и не перечисление} \\ \langle e, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = (v, \tau)\end{array}}{\langle storage\ \tau\ \texttt{id = } e\texttt{;}, \sigma\rangle \to_s \langle \epsilon, \sigma'\rangle}, \tag{38}$$

$$\begin{aligned}
\text{где}\ \ \sigma' \ &= \ \sigma(\mathrm{ID}^i \leftarrow \mathrm{ID}^i \cup \{\texttt{id}\})(\mathrm{Adr}^i \leftarrow \mathrm{Adr}^i \cup \{nc\})(\mathrm{MeM}(\texttt{id} \leftarrow nc)) \\
&\quad (\Gamma^i(\texttt{id} \leftarrow \tau))(\mathcal{I}nit(\texttt{id}, storage, v))(\mathrm{Val} := \mathrm{OkVal}), \\
\text{а}\quad i \ &= \ \begin{cases} 0, & \text{если } storage \equiv \mathsf{static}, \\ \mathrm{GLF}_\sigma, & \text{если } storage \equiv \mathsf{auto}. \end{cases}
\end{aligned}$$

Универсальная функция $\mathcal{I}nit$ была предложена в [6]. Это функция с переменным числом аргументов, два первых аргумента обязательные — имя объекта и его класс памяти. Третим аргументом может быть инициализирующее значение. Если его нет, то происходит инициализация по умолчанию. Основное отличие от [6] в том, что теперь присваивание значений происходит через адрес объекта.

Массивы и указатели. В настоящее время поддерживается инициализация только одномерных массивов. Составные объекты инициализируются костантными значениями, поэтому в посылках правил нет вычислений этих значений.

$$\frac{\begin{array}{c}\tau - \text{допустимый для массива тип} \\ \langle e, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = (v, \mathsf{unsigned\ int}) \quad v \geq 0\end{array}}{\langle storage\ \tau\ \texttt{id[}e\texttt{];}, \sigma\rangle \to_s \langle \epsilon, \sigma''\rangle}, \tag{39}$$

$$\begin{aligned}
\text{где}\ \ \sigma'' \ &= \ \sigma'\,(\mathrm{ID}^i \leftarrow \mathrm{ID}^i \cup \{\texttt{id}\})(\mathrm{Adr}^i \leftarrow \mathrm{Adr}^i \cup \{\text{"}nc+0\text{"}, ..., \text{"}nc+v\text{"}\}) \\
&\quad (\mathrm{MeM}(\texttt{id} \leftarrow nc))(\Gamma^i(\texttt{id} \leftarrow \tau[v])) \\
&\quad (\mathcal{I}nit(\texttt{id}, storage))(\mathrm{Val} := \mathrm{OkVal}),
\end{aligned}$$

$$\text{a} \quad i \quad = \begin{cases} 0, & \text{если } storage \equiv \mathsf{static}, \\ \mathrm{GLF}_\sigma, & \text{если } storage \equiv \mathsf{auto}. \end{cases}$$

$$\frac{\begin{array}{c}\tau - \text{допустимый для массива тип} \\ \langle e, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = (v, \mathsf{unsigned\ int}) \quad v \geq 0\end{array}}{\langle storage\ \tau\ \texttt{id[}e\texttt{] = \{}init\_list\texttt{\};}, \sigma\rangle \to_s \langle \epsilon, \sigma''\rangle}, \tag{40}$$

где $\sigma'' = \sigma'\,(\mathrm{ID}^i \leftarrow \mathrm{ID}^i \cup \{\texttt{id}\})(\mathrm{Adr}^i \leftarrow \mathrm{Adr}^i \cup \{nc+0, ..., nc+v\})$
$(\mathrm{MeM}(\texttt{id} \leftarrow nc))(\Gamma^i(\texttt{id} \leftarrow \tau[v]))$
$(\mathcal{I}nit(\texttt{id}, storage, init\_list))(\mathrm{Val} := \mathrm{OkVal}),$

$$\text{a} \quad i \quad = \begin{cases} 0, & \text{если } storage \equiv \mathsf{static}, \\ \mathrm{GLF}_\sigma, & \text{если } storage \equiv \mathsf{auto}. \end{cases}$$

$$\langle storage\ \tau\ \texttt{id[}e\texttt{]}\ dimensions\texttt{;}, \sigma\rangle \to_s \langle S, \sigma\rangle, \tag{41}$$

где $S \equiv \texttt{typedef}\ \tau\ \texttt{new\_id}\,dimensions\texttt{;}\ storage\ \texttt{new\_id*\ id[}e\texttt{];}$
а `new_id` — новый уникальный идентификатор. Т.е. декларация многомерного массива разбивается на последовательность деклараций одномерных.

Инициализатором указателя, в отличие от массива, может быть неконстантное выражение, поэтому во втором правиле для указателей есть вычисление в посылке.

$$\langle storage\ \tau\texttt{*\ id;}, \sigma\rangle \to_s \langle \epsilon, \sigma'\rangle, \tag{42}$$

где $\sigma' = \sigma(\mathrm{ID}^i \leftarrow \mathrm{ID}^i \cup \{\texttt{id}\})(\mathrm{Adr}^i \leftarrow \mathrm{Adr}^i \cup \{nc\})(\mathrm{MeM}(\texttt{id} \leftarrow nc))$
$(\Gamma^i(\texttt{id} \leftarrow \tau*))(\mathcal{I}nit(\texttt{id}, storage))(\mathrm{Val} := \mathrm{OkVal}),$

$$\text{a} \quad i \quad = \begin{cases} 0, & \text{если } storage \equiv \mathsf{static}, \\ \mathrm{GLF}_\sigma, & \text{если } storage \equiv \mathsf{auto}. \end{cases}$$

$$\frac{\langle e, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = (v, \tau*)}{\langle storage\ \tau\texttt{*\ id = }e\texttt{;}, \sigma\rangle \to_s \langle \epsilon, \sigma'\rangle}, \tag{43}$$

где $\sigma' = \sigma(\mathrm{ID}^i \leftarrow \mathrm{ID}^i \cup \{\texttt{id}\})(\mathrm{Adr}^i \leftarrow \mathrm{Adr}^i \cup \{nc\})(\mathrm{MeM}(\texttt{id} \leftarrow nc))$
$(\Gamma^i(\texttt{id} \leftarrow \tau*))(\mathcal{I}nit(\texttt{id}, storage, v))(\mathrm{Val} := \mathrm{OkVal}),$

$$\text{a} \quad i \quad = \begin{cases} 0, & \text{если } storage \equiv \mathsf{static}, \\ \mathrm{GLF}_\sigma, & \text{если } storage \equiv \mathsf{auto}. \end{cases}$$

Структуры и перечисления. Фрагмент, в котором одновременно объявляются структурный тип и объекты этого типа, разбивается на отдельные декларации типа и объектов.

$$\langle \texttt{struct tag;}, \sigma\rangle \to_s \langle \epsilon, \sigma(\Sigma(\texttt{tag} \leftarrow \emptyset))\rangle. \tag{44}$$

$$\langle \texttt{struct tag\{}\tau_1\ f_1\texttt{;}...\tau_n\ f_n\texttt{;\};}, \sigma\rangle \to_s \langle \epsilon, \sigma'\rangle, \tag{45}$$

где $\sigma' = \sigma(\Sigma(\texttt{tag} \leftarrow [(f_1, \mathrm{TD}(\tau_1)), ..., (f_n, \mathrm{TD}(\tau_n))]))$.

$$\langle \texttt{struct tag\{}fields\texttt{\}}\ declarator\_list\texttt{;}, \sigma\rangle \to_s \langle S, \sigma\rangle, \tag{46}$$

где $S \equiv$ `struct tag{`*fields*`}; struct tag` *declarator_ list*`;` .

$$\langle \texttt{struct \{}fields\texttt{\}}\ declarator\_list\texttt{;}, \sigma\rangle \to_s \langle S, \sigma\rangle, \tag{47}$$

где $S \equiv$ `struct new_tag{`*fields*`}; struct new_tag` *declarator_ list*`;`,
а `new_tag` — новый уникальный идентификатор. Т.е. если у структурного типа нет имени, то заводится новое имя, чтобы отображение Σ имело смысл в данном случае.

$$\langle storage\ \texttt{struct tag id;}, \sigma\rangle \to_s \langle \epsilon, \sigma'\rangle, \tag{48}$$

где $\sigma' = \sigma(\mathrm{ID}^i \leftarrow \mathrm{ID}^i \cup \{\texttt{id}\})(\mathrm{Adr}^i \leftarrow \mathrm{Adr}^i \cup \{nc\})(\mathrm{MeM}(\texttt{id} \leftarrow nc))$
$(\Gamma^i(\texttt{id} \leftarrow \mathsf{struct}\{\Sigma^2(\texttt{id})\}))(\mathcal{I}nit(\texttt{id}, storage))(\mathrm{Val} := \mathrm{OkVal})$,

а $i = \begin{cases} 0, & \text{если } storage \equiv \mathsf{static}, \\ \mathrm{GLF}_\sigma, & \text{если } storage \equiv \mathsf{auto}. \end{cases}$

$$\langle storage\ \texttt{struct tag id = \{}init\_list\texttt{\};}, \sigma\rangle \to_s \langle \epsilon, \sigma'\rangle, \tag{49}$$

где $\sigma' = \sigma(\mathrm{ID}^i \leftarrow \mathrm{ID}^i \cup \{\texttt{id}\})(\mathrm{Adr}^i \leftarrow \mathrm{Adr}^i \cup \{nc\})$
$(\mathrm{MeM}(\texttt{id} \leftarrow nc))(\Gamma^i(\texttt{id} \leftarrow \mathsf{struct}\{\Sigma_2(\texttt{id})\}))$
$(\mathcal{I}nit(\texttt{id}, storage, init\_list))(\mathrm{Val} := \mathrm{OkVal})$,

а $i = \begin{cases} 0, & \text{если } storage \equiv \mathsf{static}, \\ \mathrm{GLF}_\sigma, & \text{если } storage \equiv \mathsf{auto}. \end{cases}$

Объявление любого перечисления трактуется просто как одновременное объявление нескольких переменных типа signed int.

$$\langle \texttt{enum tag\{}drator\_list\texttt{\};}, \sigma\rangle \to_s \langle \texttt{signed int}\ drator\_list\texttt{;}, \sigma\rangle. \tag{50}$$

$$\langle \texttt{enum tag\{}drator\_list_1\texttt{\}}drator\_list_2\texttt{;}, \sigma\rangle \to_s \langle S, \sigma\rangle, \tag{51}$$

где $S \equiv$ `signed int` $drator\_list_1$`,`$drator\_list_2$`;` .

Функции.

$$\frac{\mathsf{id} \neq \mathsf{main} \quad B - \text{блок или пустой оператор}}{\langle \tau\ \mathsf{id}(\tau_1\ v_1, ..., \tau\ v_n)\, B, \sigma\rangle \to_s \langle \epsilon, \sigma'\rangle}, \tag{52}$$

где $\sigma' = \sigma(\mathrm{ID}^0 \leftarrow \mathrm{ID}^0 \cup \{\mathsf{id}\})$
$(\Gamma^0(\mathsf{id} \leftarrow (\mathrm{TD}(\tau_1) \times ... \times \mathrm{TD}(\tau_n) \to \mathrm{TD}(\tau))))$
$(\mathrm{Val} := \mathrm{OkVal})$.

Т.е. прототип и определение функции обрабатываются одним правилом. Вследствие запрета указателей на функции вводится только имя с типом, а адрес и значение отсутствуют. Можно также считать, что с именем функции неявно связывается список деклараций ее аргументов и ее тело, информация о которых будет использоваться при вызове. Заметим, что правило для функции main дано отдельно в следующем пункте.

2.7. Семантика операторов

Для краткости мы полагаем следующее. Если в посылке правила не записаны никакие предположения о компоненте Val_σ, где σ — то состояние, с которого начинается исполнение рассматриваемого оператора, то есть стандартная неявная посылка: $\mathrm{Val}_\sigma = \mathrm{OkVal}$. Поскольку значение OkVal соответствует «нормальной» работе программы, легко заметить, что эта посылка эквивалентна следующей:

$$\mathrm{Val}_\sigma \neq (\mathrm{GoVal} \vee \mathrm{BreakVal} \vee \mathrm{ContVal} \vee \mathrm{RetVal} \vee \mathrm{CaseVal} \vee \mathrm{Fail}).$$

Если же в правиле требуется другая посылка, она будет записываться явно.

Пустой оператор.

$$\langle ;, \sigma\rangle \to_s \langle \epsilon, \sigma\rangle. \tag{53}$$

Оператор-выражение. Конец любого полного выражения является контрольной точкой.

$$\frac{\langle e, \sigma_0\rangle \to_e^* \langle \epsilon, \sigma\rangle \qquad \mathrm{Val}_\sigma \neq \omega}{\langle e;, \sigma_0\rangle \to_s \langle \epsilon, \sigma(\mathrm{Val} := \mathrm{OkVal})\rangle}. \tag{54}$$

$$\frac{\langle e, \sigma_0\rangle \to_e^* \langle \epsilon, \sigma\rangle \qquad \mathrm{Val}_\sigma = \omega}{\langle e;, \sigma_0\rangle \to_s \langle \epsilon, \sigma(\mathrm{Val} := \mathrm{Fail})\rangle}. \tag{55}$$

Помеченный оператор. Помимо обычных меток, есть case и default-метки.

$$\frac{S - \text{оператор} \qquad \text{Val}_\sigma = \text{OkVal}}{\langle \texttt{L:}\, S, \sigma\rangle \to_s \langle S, \sigma\rangle\rangle}. \tag{56}$$

$$\frac{S - \text{оператор} \qquad \text{Val}_\sigma = \text{GoVal}(\texttt{L})}{\langle \texttt{L:}\, S, \sigma\rangle \to_s \langle S, \sigma(\text{Val} := \text{OkVal}))\rangle}. \tag{57}$$

$$\frac{S - \text{оператор} \qquad \text{Val}_\sigma = \text{GoVal}(\texttt{L}') \qquad \texttt{L} \neq \texttt{L}'}{\langle \texttt{L:}\, S, \sigma\rangle \to_s \langle \epsilon, \sigma\rangle\rangle}. \tag{58}$$

$$\frac{S - \text{оператор} \qquad \text{Val}_\sigma = \text{OkVal}}{\langle \texttt{case}\ c\texttt{:}\ \ S, \sigma\rangle \to_s \langle S, \sigma\rangle}. \tag{59}$$

$$\frac{S - \text{оператор} \qquad \text{Val}_\sigma = \text{CaseVal}(c)}{\langle \texttt{case}\ c\texttt{:}\ \ S, \sigma\rangle \to_s \langle S, \sigma(\text{Val} := \text{OkVal})\rangle}. \tag{60}$$

$$\frac{S - \text{оператор} \quad \text{Val}_\sigma = \text{BreakVal} \vee (\text{CaseVal}(c') \wedge c \neq c')}{\langle \texttt{case}\ c\texttt{:}\ \ S, \sigma\rangle \to_s \langle \epsilon, \sigma\rangle}. \tag{61}$$

$$\frac{S - \text{оператор} \qquad \text{Val}_\sigma = \text{OkVal} \vee \text{CaseVal}(..)}{\langle \texttt{default:}\ \ S, \sigma\rangle \to_s \langle S, \sigma(\text{Val} := \text{OkVal})\rangle}. \tag{62}$$

$$\frac{S - \text{оператор} \qquad \text{Val}_\sigma \neq \text{OkVal} \vee \text{CaseVal}(..)}{\langle \texttt{default:}\ \ S, \sigma\rangle \to_s \langle \epsilon, \sigma\rangle}. \tag{63}$$

Замечание. Любая посылка вида "S – оператор" означает, что S — произвольный синтаксически правильный оператор языка C-light. В п. "Циклы" будет описана конструкция **loop**(e,S), которая, как и **OrAnd**, является промежуточной, поэтому не попадает под действие этой посылки. У **loop** будут свои правила перехвата исключительных значений.

Передача управления и исключения. В предыдущих пунктах и в [5, 6] подробно рассматривалась проблема операторов передачи управления. Метод, предложенный в [16], был распространен на оператор goto, и в результате семантика всех операторов передачи управления задается единым образом. Любой такой оператор вызывает появление некоторого исключительного значения. По аналогии с механизмом обработки исключений в каких-то других конструкциях эти исключения будут перехватываться.

$$\langle \texttt{goto L;}, \sigma\rangle \to_s \langle \epsilon, \sigma(\text{Val} := \text{GoVal(L)})\rangle. \tag{64}$$

$$\langle \texttt{break;}, \sigma\rangle \to_s \langle \epsilon, \sigma(\mathrm{BreakVal})\rangle. \tag{65}$$

$$\langle \texttt{continue;}, \sigma\rangle \to_s \langle \epsilon, \sigma(\mathrm{Val} := \mathrm{ContVal})\rangle. \tag{66}$$

$$\langle \texttt{return;}, \sigma\rangle \to_s \langle \epsilon, \sigma(\mathrm{Val} := \mathrm{RetVal}(\emptyset, \mathsf{void}))\rangle. \tag{67}$$

$$\frac{\langle e, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \qquad \mathrm{Val}_{\sigma'} = (v, \tau)}{\langle \texttt{return } e\texttt{;}, \sigma\rangle \to_s \langle \epsilon, \sigma(\mathrm{Val} := \mathrm{RetVal}(v, \tau))\rangle}. \tag{68}$$

Достоинство данного подхода в том, что при передаче управления не следует сразу искать точку программы, в которую надо перейти. Этот поиск — сложно-формализуемый процесс. Гораздо проще игнорировать все последующие операторы, пока управление не попадет в нужную точку либо окажется в таком месте программы, где искомая точка определяется элементарно.

В правилах (57) и (58) уже описывался перехват значения GoVal(L) в помеченном операторе. Очевидно, что любой непомеченный оператор при этом значении должен игнорироваться, причем это верно и для остальных значений:

$$\frac{\begin{array}{c} S - \text{непомеченный оператор} \\ \mathrm{Val}_\sigma = \mathrm{GoVal}(..) \vee \mathrm{BreakVal} \vee \mathrm{ContVal} \vee \mathrm{RetVal}(..) \vee \mathrm{Fail} \end{array}}{\langle S, \sigma\rangle \to_s \langle \epsilon, \sigma\rangle}. \tag{69}$$

Ранее уже отмечалось, что в большинстве других правил есть неявная посылка OkVal, без которой семантика станет недетерминированной. Таким образом это правило «самого внешнего уровня» — прежде чем перейти к обработке конструкции по правилам семантики, необходимо убедиться, что эта конструкция вообще будет исполнена. Кроме этого правила, некоторые исключительные значения перехватываются и в других правилах.

Составные операторы. Основное правило композиции операторов имеет обычный вид:

$$\frac{S_1 - \text{оператор} \qquad \langle S_1, \sigma\rangle \to_s \langle S_2, \sigma'\rangle}{\langle S_1\, T, \sigma\rangle \to_s \langle S_2\, T, \sigma'\rangle}. \tag{70}$$

Благодаря новому подходу для блоков задаются всего два правила. Идея состоит в том, что тело цикла при передаче управления назад формируется тогда, когда оператор `goto L` и сама метка `L` находятся в

одном блоке на одном уровне вложенности. По правилам, рассмотренным ранее, генерация значения GoVal(..) не изменяет порядка вычисления, поэтому здесь возможны две ситуации. Первая соответствует тому, что в блоке либо вообще не был встречен оператор `goto L`, либо была передача и метка `L` была встречена в теле блока[3], либо была передача, но метка отсутствует в теле блока (т.е. передача управления наружу). В любом случае происходит выход в охватывающий блок, который в свою очередь будет обрабатывать значение GoVal(..), если оно есть:

$$\frac{\begin{array}{c}\langle S, \sigma(\mathrm{GLF}\ {+}{+})\rangle \to_s^* \langle \epsilon, \sigma'\rangle \\ (\mathrm{Val}_{\sigma'} \neq \mathrm{GoVal}(..)) \vee (\mathrm{Val}_{\sigma'} = \mathrm{GoVal}(\mathrm{L}) \wedge \mathrm{L} \notin \mathrm{Labels}(S))\end{array}}{\langle \{S\}, \sigma\rangle \to_s \langle \epsilon, \sigma'(\mathrm{GLF}\ {-}{-})\rangle}. \tag{71}$$

Вторая ситуация однозначно соответствует передаче управления назад в пределах блока, т.е. циклу. Очевидно, что просто подставить в данной точке тело блока нельзя, поскольку будет потеряна закрывающая фигурная скобка, а использовать правила для отдельных лексем-скобок — плохой подход. Поэтому предлагается покинуть блок и снова обработать его целиком, а чтобы информация о локальных объектах не исчезла при операции $--$, она запоминается в компоненте BC:

$$\frac{\begin{array}{c}\langle S, \sigma(\mathrm{GLF}\ {+}{+})\rangle \to_s^* \langle \epsilon, \sigma'\rangle \quad \mathrm{Val}_{\sigma'} = \mathrm{GoVal}(\mathrm{L}) \wedge \mathrm{L} \in \mathrm{Labels}(S) \\ i = \mathrm{GLF}_{\sigma'} \qquad \sigma'' = \sigma'(\mathrm{BC} := [\mathrm{ID}^i, \mathrm{Adr}^i, \mathrm{MeM}^i, \mathrm{MD}^i, \Gamma^i, \Sigma^i])\end{array}}{\langle \{S\}, \sigma\rangle \to_s \langle \{S\}, \sigma''(\mathrm{GLF}\ {-}{-})\rangle}. \tag{72}$$

Условные операторы. Правила для условного оператора if очевидны: в зависимости от значения условного выражения выбирается та или иная ветвь.

$$\frac{S \text{ не содержит на верхнем уровне слово } \mathsf{else}}{\langle \texttt{if(}e\texttt{)}\ \ S\texttt{;}, \sigma\rangle \to_s \langle \texttt{if(}e\texttt{)}\ \ S\ \ \texttt{else}\ \ \texttt{\{\}}, \sigma\rangle}. \tag{73}$$

$$\frac{\langle e, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \qquad \mathrm{Val}_{\sigma'} = \omega}{\langle \texttt{if(}e\texttt{)}\ \ S_1\ \ \texttt{else}\ \ S_2, \sigma\rangle \to_s \langle \epsilon, \sigma'(\mathrm{Val} := \mathrm{Fail})\rangle}. \tag{74}$$

$$\frac{\langle e, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \qquad \mathrm{Val}_{\sigma'} = (v, \tau) \qquad IC(\tau, \mathsf{int}) \qquad v \neq 0}{\langle \texttt{if(}e\texttt{)}\ \ S_1\ \ \texttt{else}\ \ S_2, \sigma\rangle \to_s \langle S_1, \sigma'(\mathrm{Val} := \mathrm{OkVal})\rangle}. \tag{75}$$

$$\frac{\langle e, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \qquad \mathrm{Val}_{\sigma'} = (v, \tau) \qquad IC(\tau, \mathsf{int}) \qquad v = 0}{\langle \texttt{if(}e\texttt{)}\ \ S_1\ \ \texttt{else}\ \ S_2, \sigma\rangle \to_s \langle S_2, \sigma'(\mathrm{Val} := \mathrm{OkVal})\rangle}. \tag{76}$$

[3]Т.е. это передача управления вперед в пределах блока.

Введение значения CaseVal(..) позволяет не искать нужную ветвь в операторе switch, а просто переходить к его телу.

$$\frac{\langle e,\sigma\rangle \to_e^* \langle \epsilon,\sigma'\rangle \qquad \mathrm{Val}_{\sigma'} = \omega}{\langle \texttt{switch(}e\texttt{) \{}B\texttt{\},}\sigma\rangle \to_s \langle \epsilon,\sigma'(\mathrm{Val} := \mathrm{Fail})\rangle}. \tag{77}$$

$$\frac{\begin{array}{c}\langle e,\sigma\rangle \to_e^* \langle \epsilon,\sigma_1\rangle \qquad \mathrm{Val}_{\sigma_1} = (v,\tau) \qquad IC(\tau,\mathsf{int})\\ \langle B,\sigma_1(\mathrm{GLF}\,{+}{+})(\mathrm{Val} := \mathrm{CaseVal}(v))\rangle \to_s^* \langle \epsilon,\sigma_2\rangle\\ \mathrm{Val}_{\sigma_2} \neq \mathrm{BreakVal}\end{array}}{\langle \texttt{switch(}e\texttt{) \{}B\texttt{\},}\sigma\rangle \to_s \langle \epsilon,\sigma_2(\mathrm{GLF}\,{-}{-})\rangle}. \tag{78}$$

$$\frac{\begin{array}{c}\langle e,\sigma\rangle \to_e^* \langle \epsilon,\sigma_1\rangle \qquad \mathrm{Val}_{\sigma_1} = (v,\tau) \qquad IC(\tau,\mathsf{int})\\ \langle B,\sigma_1(\mathrm{GLF}\,{+}{+})(\mathrm{Val} := \mathrm{CaseVal}(v))\rangle \to_s^* \langle \epsilon,\sigma_2\rangle\\ \mathrm{Val}_{\sigma_2} = \mathrm{BreakVal}\end{array}}{\langle \texttt{switch(}e\texttt{) \{}B\texttt{\},}\sigma\rangle \to_s \langle \epsilon,\sigma_2(\mathrm{GLF}\,{-}{-})(\mathrm{Val} := \mathrm{OkVal})\rangle}. \tag{79}$$

Последнее правило ограничивает область действия оператора break, если он сработал в теле оператора-переключателя.

Циклы. В [5] для моделирования циклов использовались конструкции **T** и **O**, предложенные в [16]. В данной версии семантики новая модель памяти, поэтому эти конструкции не подходят. Предлагается использовать новую промежуточную конструкцию $\mathbf{loop}(e,S)$, которая является наиболее общим циклом и полностью соответствует циклу while. Выражение e — это условие цикла, а S — тело цикла. Вначале опишем правила семантики для **loop**.

$$\frac{\mathrm{Val}_\sigma = \mathrm{OkVal} \vee \mathrm{ContVal} \qquad \langle e,\sigma\rangle \to_e^* \langle \epsilon,\sigma'\rangle \qquad \mathrm{Val}_{\sigma'} = \omega}{\langle \mathbf{loop}(e,S),\sigma\rangle \to_s \langle \epsilon,\sigma'(\mathrm{Val} := \mathrm{Fail})\rangle}. \tag{80}$$

$$\frac{\begin{array}{c}\mathrm{Val}_\sigma = \mathrm{OkVal} \vee \mathrm{ContVal} \qquad \langle e,\sigma\rangle \to_e^* \langle \epsilon,\sigma'\rangle\\ \mathrm{Val}_{\sigma'} = (v,\tau) \qquad IC(\tau,\mathsf{int}) \qquad v = 0\end{array}}{\langle \mathbf{loop}(e,S),\sigma\rangle \to_s \langle \epsilon,\sigma'(\mathrm{Val} := \mathrm{OkVal})\rangle}. \tag{81}$$

$$\frac{\begin{array}{c}\mathrm{Val}_\sigma = \mathrm{OkVal} \vee \mathrm{ContVal} \qquad \langle e,\sigma\rangle \to_e^* \langle \epsilon,\sigma'\rangle\\ \mathrm{Val}_{\sigma'} = (v,\tau) \qquad IC(\tau,\mathsf{int}) \qquad v \neq 0\end{array}}{\langle \mathbf{loop}(e,S),\sigma\rangle \to_s \langle S\ \mathbf{loop}(e,S),\sigma'(\mathrm{Val} := \mathrm{OkVal})\rangle}. \tag{82}$$

$$\frac{\mathrm{Val}_\sigma = \mathrm{BreakVal}}{\langle \mathbf{loop}(e,S),\sigma\rangle \to_s \langle \epsilon,\sigma(\mathrm{Val} := \mathrm{OkVal})\rangle}. \tag{83}$$

$$\frac{\mathrm{Val}_\sigma = \mathrm{GoVal(L)} \quad \mathrm{L} \in \mathrm{Labels}(S)}{\langle \mathbf{loop}(e, S), \sigma\rangle \to_s \langle S\ \mathbf{loop}(e, S), \sigma\rangle}. \tag{84}$$

$$\frac{(\mathrm{Val}_\sigma = \mathrm{RetVal(..)} \vee \mathrm{Fail}) \vee (\mathrm{Val}_\sigma = \mathrm{GoVal(L)} \wedge \mathrm{L} \notin \mathrm{Labels}(S))}{\langle \mathbf{loop}(e, S), \sigma\rangle \to_s \langle \epsilon, \sigma\rangle}. \tag{85}$$

Заметим, что значение CaseVal(..) не отлавливается, поскольку такая ситуация невозможна по определению.

Прежде чем переходить от обычных циклов к общей конструкции, необходимо убедиться, что тело цикла — это блок. Расстановка скобок не противоречит языку C, поскольку согласно стандарту любой цикл — это блок, но наличие скобок важно при переходе к **loop**.

$$\frac{S - \text{оператор и не блок}}{\langle \texttt{while(}e\texttt{)}\ S\texttt{;}, \sigma\rangle \to_s \langle \texttt{while(}e\texttt{)}\ \texttt{\{}S\texttt{\};}, \sigma\rangle}. \tag{86}$$

$$\frac{B - \text{оператор и не блок}}{\langle \texttt{for(}S\texttt{)}\ B\texttt{;}, \sigma\rangle \to_s \langle \texttt{for(}S\texttt{)}\ \texttt{\{}B\texttt{\};}, \sigma\rangle}. \tag{87}$$

$$\frac{S - \text{оператор и не блок}}{\langle \texttt{do}\ S\ \texttt{while(}e\texttt{);}, \sigma\rangle \to_s \langle \texttt{do}\ \texttt{\{}S\texttt{\}}\ \texttt{while(}e\texttt{);}, \sigma\rangle}. \tag{88}$$

И наконец переход от циклов к **loop**:

$$\langle \texttt{while(}e\texttt{)}\ \texttt{\{}S\texttt{\};}, \sigma\rangle \to_s \langle \texttt{\{}\mathbf{loop}(e, S)\texttt{\}}, \sigma\rangle. \tag{89}$$

$$\langle \texttt{do}\ \texttt{\{}S\texttt{\}}\ \texttt{while(}e\texttt{);}, \sigma\rangle \to_s \langle \texttt{\{}S\ \mathbf{loop}(e, S)\texttt{\}}, \sigma\rangle. \tag{90}$$

$$\langle \texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}\ \texttt{\{}S\texttt{\};}, \sigma\rangle \to_s \langle S', \sigma\rangle, \tag{91}$$

где $S' \equiv \texttt{\{}e_1\texttt{;}\ \texttt{if(!}e_2\texttt{)}\ \texttt{break;}\ S\ \mathbf{loop}(\texttt{(}e_3\texttt{,}e_2\texttt{)}, S)\texttt{\}}$.

Проверять условие первой итерации в последней аксиоме приходится через отрицание, поскольку иначе пришлось бы добавить фигурные скобки вокруг ветви then, что изменит семантику. Здесь важно, чтобы все содержимое цикла, как блока, образовывало один уровень вложенности. Именно для этого были введены правила (86)–(88).

Функция main. Работа программы на языке C начинается с передачи управления в функцию main, при этом явного вызова этой функции нет. Имя main зарезервировано, поэтому можно задать правило, моделирующее этот переход. Напомним, что в данной версии семантики аргументы у этой функции не поддерживаются.

$$\langle \texttt{int main(void)\{}S\texttt{\}}, \sigma\rangle \to_s \langle \texttt{\{}S\texttt{\}}, \sigma\rangle. \tag{92}$$

Как и в [5, 6], рассмотренная семантика обладает свойствами детерминизма и отсутствия блокировки. Для нее также выполняется лемма о монотонности из [6], которая используется при доказательстве корректности перевода. Формальное определение семантики частичной корректности приводится в п. 4.1.

3. ПРАВИЛА ПЕРЕВОДА ИЗ ЯЗЫКА C-LIGHT В C-LIGHT-KERNEL

В начале данного разд. приводится краткий обзор языка C-light-kernel и его основных отличий от C-light. Это позволит пояснить причины введения того или иного правила. Сами правила перевода разбиты на две непересекающиеся группы — правила для выражений и правила для операторов и деклараций. В каждой группе правила независимы, т.е. могут применяться в любом порядке, но сами группы применяются одна за другой — сперва переписываются операторы и декларации (п. 3.2), затем выражения (п. 3.3). Переписывание деклараций объединено с операторами, поскольку используемые правила имеют вид переписывания операторов.

3.1. Краткий обзор языка C-light-kernel

Язык C-light-kernel [6], вообще говоря, не является «самостоятельным» языком программирования. Это промежуточный язык двухуровневой схемы верификации программ, в который транслируются программы на языке C-light.

Синтаксис лексем языка C-light-kernel совпадает с синтаксисом лексем языка C-light [5], однако множество ключевых слов сокращается, и остаются следующие: auto, bool, char, delete, double, else, enum, false, float, goto, if, int, long, new, return, short, signed, sizeof, static, struct, true, typedef, unsigned, void, wchar_t, while.

Декларации. Списки декларируемых объектов разрешены только в декларациях функций, любая другая декларация объявляет ровно один объект.

В C-light-kernel спецификаторы класса памяти static и auto становятся обязательными, что позволяет просто задать семантику неявной инициализации.

Выражения. При вычислении любого выражения в C-light-kernel допускается не более одного изменения содержимого памяти, т.е. побочные эффекты в выражениях единичные. Допустимыми операциями являются следующие:

1) **первичные**: `()` `[]` `.` `->`
2) **унарные**: `*` `-`
3) **бинарные**: `*` `/` `%` `+` `-` `<` `>` `<=` `>=` `==` `!=`
4) **присваивание**: `=`
5) **работа с памятью**: `new` `delete`
7) **приведение типа**.

Выражение присваивания может содержать операции присваивания, а может быть простым rvalue. Не разрешаются цепочки присваиваний вида `a=b=c`. В C-light-kernel нет операции «запятая», поэтому любой оператор вычисления выражения есть либо выражение присваивания, либо вызов процедуры.

Одна из целей трансляции из языка C-light в язык C-light-kernel — устранение побочных эффектов. Поэтому фактическим аргументом функции может быть либо константа, либо переменная, но не выражение с какими-либо операциями.

Операторы. В языке C-light-kernel допустимыми являются следующие операторы.

1. Оператор вычисления выражения (возможно пустой).
2. Условный оператор if с обязательной ветвью else (возможно пустой).
3. Оператор цикла while, условием которого может быть только выражение без побочных эффектов.
4. Оператор передачи управления goto с теми же ограничениями, что и в C-light .
5. Составной оператор (блок).

3.2. Переписывание операторов и деклараций

3.2.1. Правила уточнения класса памяти

Dec1. Всякая декларация вида

type_spec declarator_list;

в которой отсутствует явная спецификация класса памяти, заменяется на декларацию вида

storage type_spec declarator_list;

где *storage* — это, в зависимости от места самой декларации, либо ключевое слово static, либо ключевое слово auto.

Dec2. Всякая декларация вида

register *type_spec declarator_list*;

заменяется на декларацию вида

auto *type_spec declarator_list*;

Наличие явных спецификаторов класса памяти позволит задать аксиоматическую семантику деклараций языка C-light-kernel без проверки уровней вложенности.

3.2.2. Правило разбиения списка деклараторов

Dec3. Всякая декларация вида

decl_specifiers declarator, declarator_list;

заменяется на фрагмент вида

decl_specifiers declarator;
decl_specifiers declarator_list;

В результате в каждой декларации объявляется ровно один объект.

3.2.3. Правила нормализации

Правила нормализации приводят операторы к виду, удобному для применения правил из остальных групп. Расставляются дополнительные фигурные скобки; условием операторов if становится значение переменной, а не выражения; сами операторы выбора получают недостающие ветви else и default; условием оператора while становится константа 1 (т.е. логическая истина).

Norm1. Фрагмент вида

```
while(e) A
```

где A — оператор, не являющийся блоком, заменяется на

```
while(e){A}
```

Norm2. Фрагмент вида

```
do A while(e)
```

где `A` — оператор, не являющийся блоком, заменяется на

```
do {A} while(e)
```

Norm3. Фрагмент вида

```
for(A) B
```

где `B` — оператор, не являющийся блоком, заменяется на

```
for(A){B}
```

Norm4. Фрагмент вида

```
if(e) A B
```

где `A` — инструкция, `B` не начинается с `else`, заменяется на

```
if(e) A else {}; B
```

Norm5. Фрагмент вида

```
switch(e){A}
```

где `A` не содержит `default`, заменяется на

```
switch(e){A default: {}}
```

Norm6. Фрагмент вида

```
if(e) A
```

где `e` не переменная, заменяется на

```
{auto t x; x = e; if(x) A}
```

где `x` — новая переменная, `t` — тип выражения `e`;

Norm7. Фрагмент вида

```
switch(e){A}
```

где `e` не переменная, заменяется на

```
{auto t x; x = e; switch(x) {A}}
```

где `x` — новая переменная, `t` — тип выражения `e`;

Norm8. Фрагмент вида

```
while(e){A}
```

где `A` не содержит `continue`, заменяется на

```
while(1){if(e){} else break; A}
```

3.2.4. Правила элиминации операторов for, do-while и switch

LSE1. Фрагмент вида

```
switch(x) {A case v: B default: C}
```

где `x` — переменная, `A` содержит `case`, `B` не содержит `case` на верхнем уровне, заменяется на

```
switch(x) {A default: if(x == v) {B} C}
```

LSE2. Фрагмент вида

```
switch(x) {case v: B default: C}
```

где `x` — переменная, `B` не содержит `case` на верхнем уровне, заменяется на

```
switch(x) {default: if(x == v) {B} C}
```

LSE3. Фрагмент вида

```
switch(x) {default: A case v: B}
```

где `x` — переменная, `A` не содержит `case` на верхнем уровне, заменяется на

```
switch(x) {default: {if(x == v) goto l; A l: B}}
```

где `l` — новая метка;

LSE4. Фрагмент вида

```
switch(x) {default: A}
```

где `A` не содержит `case` и `break`, заменяется на

```
{A}
```

LSE5. Фрагмент вида

```
do {A} while(e)
```

где `A` не содержит `continue`, заменяется на

```
while(1) {A if (e) {} else break;}
```

LSE6. Фрагмент вида

```
for(e1; e2; e3) {A}
```

где `A` не содержит `continue`, заменяется на

```
{e1; while(e2) {A e3}}
```

Чтобы не вводить в аксиоматической семантике перехват исключений операторы циклов выражаются через оператор while, а операторы передачи управления — через оператор goto (см. далее).

Аналогично оператор-переключатель switch преобразуется к условному оператору собиранием ветвей и расстановкой операторов break.

3.2.5. Правила элиминации операторов continue и break

ВСЕ1. Фрагмент вида

```
switch(e) {A break; B}
```

заменяется на

```
{switch(e){A goto l; B} l:}
```

где `l` — новая метка;

ВСЕ2. Фрагмент вида

```
while(e) {A break; B}
```

заменяется на

```
{while(e) {A goto l; B} l:}
```

где `l` — новая метка;

ВСЕ3. Фрагмент вида

```
while(e) {A continue; B}
```

заменяется на

```
while(e) {A goto l; B l:}
```

где `l` — новая метка;

ВСЕ4. Фрагмент вида

```
do {A continue; B} while(e)
```

заменяется на

```
do {A goto l; B l:} while(e)
```

где `l` — новая метка;

ВСЕ5. Фрагмент вида

```
for(A) {A continue; B}
```

заменяется на

```
for(A) {A goto l; B l:}
```

где `l` — новая метка.

3.3. Переписывание выражений

Переписывание выражений сводится к замене каждого выражения, в котором содержатся несколько «подозрительных» подвыражений, на цепочку последовательно вычисляемых выражений, в которой эти подвыражения оказываются вынесенными на верхний уровень. К «подозрительным» будем относить выражения с возможными побочными эф-

фектами либо выражения, в которых не все операнды вычисляются.

3.3.1. Правила элиминации логических операций

Logic1. Фрагмент вида `e||e′` заменяется на `(e?1:e′)`

Logic2. Фрагмент вида `e&&e′` заменяется на `(e?e′:0)`

Logic3. Фрагмент вида `!e` заменяется на `(e?0:1)`

При описании семантики таких операций, как логическое И и ИЛИ, обсуждалась их сложность, которая приводила к промежуточной форме **OrAnd** с большим числом правил. Поэтому эти операции заменяются условной операцией, которая, в свою очередь, тоже будет переписана. Для однородности выражений логическое отрицание также переписывается. Важно отметить, что согласно новому стандарту языка C [3] логическая истина — это всегда единица, а не произвольное ненулевое значение.

3.3.2. Правило элиминации операции выборки из массива

Ind. Фрагмент вида `a[i]` заменяется на `*(a+i)`

3.3.3. Правила элиминации операций инкремента и декремента

Inc1. Фрагмент вида `++e` заменяется на `(e += 1)`

Inc2. Фрагмент вида `e++` заменяется на

```
(q = &e, y = *q, *q = *q + 1, y)
```

где `q`, `y` — новые переменные, объявленные с типами выражений `&e` и `e`, соответственно.

Отметим, что при переписывании постфиксного инкремента значение выражения `e` сохраняется не в простой переменной, а с помощью указателя, чтобы возможные побочные эффекты не повторялись. Для операций декремента знак + заменяется на −.

3.3.4. Правило элиминации составных присваиваний

CAs. Фрагмент вида

```
e op e′
```

где `op` — операция составного присваивания, заменяется на

```
(x = e′, y = &e, *y = *y op′ x)
```

где x и y — новые переменные, объявленные с типами выражений e' и &e соответственно, а op' есть +, если op есть += и т.д.

3.3.5. Правило декомпозиции выражений

Ops. Фрагмент вида

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

где e_i не переменная и не константа, $e_{i+1}, \ldots, e_n$ — переменные или константы, f — функция или одна из операций +, -, *, /, <, >, <=, =>, !=, == заменяется на

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

где x — новая переменная, объявленная с типом выражения e_i.

Жесткое ограничение на вид аргументов e_i требуется для того, чтобы порядок переписываний был задан строго однозначно.

3.3.6. Правило нормализации левой части операции присваивания

LHS. Фрагмент вида

$$e = e'$$

где e не имеет вид z или *z для некоторой переменной z, заменяется на

$$(x = e', y = \&e, *y = x)$$

где x, y — новые переменные, объявленные с типами выражений e' и &e, соответственно.

3.3.7. Правила нормализации операции взятия адреса

Adr1. Фрагмент вида *&e заменяется на e

Adr2. Фрагмент вида &*e заменяется на e

3.3.8. Правило элиминации условной операции

Tern. Фрагмент вида

```
e1?e2:e3;
```

заменяется на

```
{x = e1; if(x) {e2;} else {e3;}}
```

Условная операция может вычисляться не полностью (т.е. контрольная точка), поэтому ее заменяем на условный оператор.

3.3.9. Правила нормализации правой части операции присваивания

RHS1. Фрагмент вида

$$\texttt{x = e?}e_1\texttt{:}e_2$$

где `x` имеет вид `z` или `*z` для некоторой переменной `z`, заменяется на

$$\texttt{e?(x = }e_1\texttt{):(x = }e_2\texttt{)}$$

RHS2. Фрагмент вида

$$\texttt{x = }e_1\texttt{, ..., }e_n$$

где `x` имеет вид `z` или `*z` для некоторой переменной `z`, заменяется на

$$\texttt{(}e_1\texttt{, ..., x = }e_n\texttt{)}$$

Для корректной обработки ситуации, когда значение условной операции является операндом операции присваивания, это присваивание должно просачиваться в условный оператор.

3.3.10. Правило элиминации операции запятая

Comma. Фрагмент вида e_1`, ...,` e_n`;` заменяется на `{`e_1`; ...,` e_n`;}`

Цепочки выражений, разрастающиеся при переписывании выражений, становятся цепочками операторов вычисления выражений.

3.3.11. Правило нормализации скобок

Pars. Фрагмент вида `(e)` заменяется на `e`, если это не аргумент вызова функции и не операнд какой-либо неодноместной операции.

В заключение отметим, что в результате перевода синтаксис выражений в выходной программе соответствует некоторой трехадресной машине — в любом операторе-выражении, кроме операции присваивания, может быть не более одной двухместной операции. Тем самым побочные эффекты становятся единичными: в одном полном выражении содержимое памяти может быть изменено не более одного раза. Все выражения вычисляются полностью. Набор операторов, остающихся в языке C-light-kernel, семантически полностью соответствует аналогичным операторам языка Паскаль. Формальное доказательство свойств корректности и завершимости процесса перевода проводится в следующем разделе.

4. ОБОСНОВАНИЕ КОРРЕКТНОСТИ ПЕРЕВОДА

Формальное доказательство корректности правил перевода разбито на три части. Основным стало обоснование того, что правила перевода не изменяют семантику программы. Для этого было разработано понятие семантического расширения, свойства которого рассмотрены в п. 4.1. Доказательство теоремы о связи правил с семантическим расширением приводится в п. 4.2. Доказательству завершимости процесса перевода посвящен п. 4.3. И наконец, помимо семантической корректности, необходимо доказать «синтаксическую» корректность правил. Соответствующее утверждение, обычно называемое теоремой о нормальной форме, доказывается в п. 4.4.

4.1. Предварительные замечания и определения

Для того чтобы доказать сохранение семантики программы при переводе, необходимо задать семантику не только отдельных конструкций, но и программы целиком. По аналогии с [5, 6, 8], определяя транзитивное рефлексивное замыкание ($\rightarrow_s^*$) для отношения перехода $\rightarrow_s$, можем определить семантику программы S как отображение из *Spaces* в 2^{Spaces}:

$$\mathcal{M}[\![S]\!](\sigma) = \{\tau \,|\langle S, \sigma\rangle \rightarrow_s^* \langle \epsilon, \tau\rangle\}.$$

Т. е. семантика рассматривает все возможные исполнения программы, начинающиеся из состояния σ. Отображение $\mathcal{M}$ называется ***семантикой частичной корректности*** для языка C-light. Множество $\mathcal{M}[\![S]\!](\sigma)$ всегда одноэлементное, так как семантика языка C-light детерминированная [6]. Также по определению, если P — формула, то:

$$\mathcal{M}[\![S]\!](\|P\|) = \bigcup_{\sigma \in \|P\|} \mathcal{M}[\![S]\!](\sigma).$$

Определение. *Программа* на языке C-light — это либо последовательность внешних деклараций с одной выделенной функцией — main, либо определение одной функции.

Согласно правилам языка C, и следовательно C-light, работа программы начинается с передачи загрузчиком управления функции main и заканчивается выходом из нее при отсутствии аварийного останова. Формальная семантика должна следовать этому правилу, поэтому верифицируемая полноценная программа должна иметь именно такой вид.

Но вообще говоря, с точки зрения операционной семантики, входной программой может быть в принципе что угодно: один оператор, одно выражение либо несколько операторов. Часто необходимо верифицировать не полную программу, а некоторую функцию саму по себе. Для того чтобы не приходилось вводить в такой ситуации функцию main, понятие допустимой программы расширяется отдельными функциями.

Определение. Множеством *семантических* объектов программы S на языке C-light называется множество $\mathbf{Obj}(S)$ идентификаторов функций и всех статических переменных этой программы.

Определение. Множеством *семантических* объектов функции f называется множество $\mathbf{Obj}(f)$ идентификаторов автоматических и статических переменных функции f.

Определение. Пусть id — элемент множества семантических объектов. Тогда *атрибут* идентификатора id определяется следующим образом:

$$\mathbf{Attrib}(id) = \begin{cases} (\text{type, storage}), & \text{если } id \text{ – переменная типа type} \\ & \text{с классом памяти storage,} \\ (\tau_{Ret}, \tau_1, ..., \tau_n), & \text{если } id \text{ – имя функции типа} \\ & \tau_1 \times ... \times \tau_n \to \tau_{Ret}. \end{cases}$$

Определение. Пусть S и T — программы (функции) на языке C-light, а ϕ — некоторое взаимно-однозначное всюду определенное отображение из $\mathbf{Obj}(S)$ в $\mathbf{Obj}(T)$. Программа (функция) T называется *объектным расширением* программы (функции) S *относительно отображения* ϕ, если для любого $id \in \mathbf{Obj}(S)$

$$\mathbf{Attrib}(id) = \mathbf{Attrib}(\phi(id)). \qquad (*)$$

Это означает, что множество объектов программы (функции) T было получено переименовыванием объектов программы (функции) S и расширением какими-то новыми объектами. Отображение ϕ может быть и тождественным. Отношение объектного расширения относительно ϕ записываем как $S \lhd_\phi T$.

Важным свойством этого определения является то, что отображение не обязательно связывает только одноименные объекты. Тем самым можно сравнивать программы, полученные не только добавлением

новых объектов, но и переименовыванием старых. Хотя предложенная система правил не переименовывает переменные, а только вводит новые, при переходе к языку C++ такая ситуация может возникнуть. Очевидно, что существование такого отображения еще не означает, что программы реализуют одну и ту же вычислимую функцию. Более того, отображений, удовлетворяющих свойству $(*)$, может быть несколько. Но если среди таких отображений есть хотя бы одно такое, что при любых входных данных результаты работ программ S и T «совпадают на объектах», связанных этой инъекцией, то относительно этих объектов программы эквивалентны. Определим формально такую локальную «эквивалентность». Далее для удобства слово программа используется для обозначения обычных программ и функций.

Определение. Программа T называется *семантическим расширением* программы S, если существует всюду определенное взаимно-однозначное отображение $\phi : \mathbf{Obj}(S) \to \mathbf{Obj}(T)$ такое, что

1) $S \lhd_\phi T$;
2) для любого состояния σ

$$\phi(\mathcal{M}[\![S]\!](\sigma)) = \mathcal{M}[\![T]\!](\sigma)\Big|_{\phi(\mathbf{Obj}(S))};$$

3) для каждой пары функций $f_1 \in S$ и $f_2 \in T$ таких, что $\phi(f_1) = f_2$, существует всюду определенное взаимно-однозначное отображение $\phi_{(f_1,f_2)} : \mathbf{Obj}(f_1) \cup \mathbf{Obj}(S) \to \mathbf{Obj}(f_2) \cup \mathbf{Obj}(T)$ такое, что
   (a) $f_1 \lhd_{\phi_{(f_1,f_2)}} f_2$,
   (b) $\phi_{(f_1,f_2)}\Big|_{\mathbf{Obj}(S)} = \phi$,
   (c) для любого состояния σ

$$\phi_{(f_1,f_2)}(\mathcal{M}[\![f_1]\!](\sigma)) = \mathcal{M}[\![f_2]\!](\sigma)\Big|_{\phi_{(f_1,f_2)}(\mathbf{Obj}(f_1)\cup\mathbf{Obj}(S))},$$

где применение ϕ к состоянию означает переименовывание соответствующих идентификаторов. Равенство состояний понимается как покомпонентное совпадение. Вертикальная черта, как обычно, означает сужение на множество объектов.

Рассмотрим это определение подробнее. Пункт 1 означает, что программа T объектно расширяет S, т.е. ϕ — связывание объектов на глобальном уровне. Пункт 2 означает, что результаты выполнения программ на семантических объектах, связанных отображением ϕ, совпадают. В случае, когда S и T — функции, этого пункта достаточно для

утверждения, что они «эквивалентны». Однако для обычных программ рассматривать окончательные результаты их исполнений не достаточно. Ведь в этом случае рассматриваться будут только статические объекты, а любые автоматические при выходе из блока (и соответственно из функции main) просто исчезают. Тогда возможна следующая ситуация: в программах S и T есть одинаковые статические объекты, которые вообще не изменяются в ходе работы (т.е. используются как константы), но относительно своих автоматических объектов программы реализуют разные вычислимые функции. Но в конце вычислений сравнить можно будет только статические объекты и программы будут эквивалентны. Поэтому необходимо рассматривать и некоторые промежуточные результаты. Пункт 3 делает это определение индуктивным. В нем говорится, что надо рассмотреть и все пары функций, которые отождествляются при ϕ на глобальном уровне. Здесь важно то, что к множествам семантических объектов функций добавляются глобальные множества. Ведь все функции в программе вызываются не в произвольных условиях, а в контексте, формируемом глобальными декларациями программы. Естественно, что сужение внутренней инъекции для функций f_1 и f_2 на множество $\mathbf{Obj}(S)$ должно совпадать с ϕ.

Отношение семантического расширения обозначаем как $S \preccurlyeq_{op} T$. Оно рефлексивно и транзитивно (т.е. предпорядок). Свойства симметричности нет, более того, любая попытка сделать это отношение симметричным приведет к нарушению свойства транзитивности, без которого процесс перевода некорректен. Основная причина в том, что в определении предполагается, что семантические множества второй программы всегда шире соответствующих множеств из первой. Однако специфика правил перевода такова, что добавляются только автоматические объекты. Поэтому если считать, что множества статических объектов обеих программ равномощны, и не рассматривать в заключительных состояниях функций локальные объекты, то появится симметричность, а отношение станет эквивалентностью.

Обозначим множество статических объектов произвольной функции f как $\mathbf{Sobj}(f)$.

Определение. Программу T называем *семантически эквивалентной* программе S и обозначаем это как $S \simeq_{op} T$, если существует всюду определенная биекция $\phi : \mathbf{Obj}(S) \to \mathbf{Obj}(T)$ такая, что

1) $S \lhd_\phi T$;

2) $\forall \sigma \ \phi(\mathcal{M}[\![S]\!](\sigma)) = \mathcal{M}[\![T]\!](\sigma)\Big|_{\phi(\mathbf{Obj}(S))}$;

3) для каждой пары функций $f_1 \in S$ и $f_2 \in T$ таких, что $\phi(f_1) = f_2$, существует всюду определенная биекция

$$\phi_{(f_1,f_2)} : \mathbf{Sobj}(f_1) \cup \mathbf{Obj}(S) \to \mathbf{Sobj}(f_2) \cup \mathbf{Obj}(T)$$

такая, что

(a) $f_1 \lhd_{\phi_{(f_1,f_2)}} f_2$,

(b) $\phi_{(f_1,f_2)}\Big|_{\mathbf{Obj}(S)} = \phi$,

(c) $\forall \sigma\ \phi_{(f_1,f_2)}(\mathcal{M}[\![f_1]\!](\sigma)) = \mathcal{M}[\![f_2]\!](\sigma)\Big|_{\phi_{(f_1,f_2)}(\mathbf{Sobj}(f_1)\cup\mathbf{Obj}(S))}$.

Очевидно это отношение эквивалентности:

1) $S \simeq_{op} S$ при тождественном отображении ϕ;

2) если $S \simeq_{op} T$ при некотором ϕ, то $T \simeq_{op} S$ при ϕ^{-1}, которое является всюду определенной биекцией, поскольку таково исходное ϕ;

3) если $S \simeq_{op} T$ при некотором ϕ, а $T \simeq_{op} U$ при некотором ψ, то $S \simeq_{op} U$ при $\chi = \psi \circ \phi$.

Таким образом, отношение $\preccurlyeq_{op}$ более сильное, чем $\simeq_{op}$, поскольку оно учитывает все объекты программы. Поскольку многие правила перевода вводят новые объекты, доказательство теоремы о корректности будет проводиться для этого отношения. При этом в каждом конкретном случае будет показано выполнение условий из второго определения. Поэтому в качестве следствия мы докажем, что процесс перевода сохраняет эквивалентность.

В заключение введем специальное обозначение, которое будем часто использовать при доказательстве теоремы о корректности. Пусть $Slice = [S_1, S_2, S_3, S_4, S_5, S_6]$ — это кортеж, элементы которого определяются так же, как элементы компоненты BC (стр. 17.) Пусть σ — произвольное состояние, а $\mathrm{GLF}_\sigma = i$. Тогда

$$\sigma(\mathbf{Upd}(Slice)) \tag{93}$$

обозначает состояние σ', которое отличается от σ только тем, что $\mathrm{ID}^i = S_1$, $\mathrm{Adr}^i = S_2$, $\mathrm{MeM}^i = S_3$, $\mathrm{MD}^i = S_4$, $\Gamma^i = S_5$, $\Sigma^i = S_6$.

4.2. Теорема о корректности перевода

Теорема 1. Для любого правила перевода R и любой программы S на языке C-light $S \preccurlyeq_{op} R(S)$.

Прежде чем перейти к непосредственному доказательству, рассмотрим ряд важных замечаний, которые будут использоваться далее.

1. Основной вопрос — что брать в качестве переписываемого фрагмента? Все правила имеют локальный характер. Т. е., если правило встречает в программе шаблон, для которого оно определено, то оно тут же заменяет этот шаблон на некоторый фрагмент без учета того, где находится этот шаблон и есть ли другие вхождения этого шаблона в программу. Любой шаблон — это известный фрагмент. Его можно *формально* исполнить в семантике языка C-light и тем самым узнать результат работы. Если же рассмотреть произвольную программу S, в которую входит этот шаблон, рассуждения станут неформальными. Однако, как уже отмечалось, свойство монотонности семантики, определенное в [6], сохраняется. Пусть например правило R заменяет шаблон E на фрагмент E', а программа S имеет вид $A\ E\ B$. Тогда $R(S) = R(A)\ E'\ R(B)$. Если при этом шаблон E не входит во фрагменты A и B, то результатом переписывания будет $A\ E'\ B$. Семантика фрагментов A и B не изменяется. Но тогда, если $E \preccurlyeq_{op} E'$, то по свойству монотонности $A\ E\ B \preccurlyeq_{op} A\ E'\ B$. Поэтому в качестве S из утверждения теоремы можно брать не произвольную программу, а тот шаблон, для которого определено рассматриваемое правило.

2. При доказательстве правила семантики языка C-light, в которых появляется неопределенность, не рассматриваются, так как, если исходный фрагмент приводил к неопределенности, то и преобразованный по любому правилу будет приводить к неопределенности. Т.е. для неправильных программ все правила можно считать корректными. Это соответствует тому, как в теории алгоритмов понимается эквивалентность двух абстрактных машин: они либо обе останавливаются с одинаковыми результатами, либо обе зацикливаются. Далее в доказательстве рассматривается только правильное поведение программ (без правил обработки ошибок). Если рассматриваемая программа-шаблон — это фрагмент, в котором используются некоторые переменные, но нет деклараций этих переменных, то возможны две ситуации: 1) в начальном состоянии содержится информация не о всех переменных (имена, адреса, типы и т.п.), тогда возникнет неопределенность и исходная программа-шаблон будет эквивалентна новой; 2) в начальном состоянии содержится вся необходимая для исполнения информация. Тогда, если соответствующее правило перевода корректно, то результаты будут совпадать на исходных объектах. Тем самым эквивалентность результатов будет вне зависимости от начального состояния, что и требуется по определению. Поэтому далее считаем, что начальное состояние всегда «хоро-

шее», т.е. не приводит к аварийному завершению исходной программы.

Итак, основная идея проведения доказательства состоит в следующем. Проверяется совпадение результатов при произвольном начальном состоянии, но в рассуждениях можно считать, что состояние содержит корректную информацию об объектах фрагмента, если их деклараций в нем нет. Там, где тип объектов не важен, будем для удобства считать, что это один из базовых типов, например int. Текущее проверяемое правило перевода будем обозначать символом R. Чтобы указать применяемое правило операционной семантики (там где это важно), будем писать его номер над стрелкой соответствующего отношения перехода: $\xrightarrow{(88)}_s$.

4.2.1. Корректность правил уточнения класса памяти и декомпозиции

Корректность этих правил очевидно следует из того, что они фактически реализуют три правила операционной семантики.

1. Пусть $S \equiv$ *type_ spec declarator_ list*;, а σ — произвольное состояние. Тогда

$$\langle S, \sigma\rangle \xrightarrow{(34)}_s \langle \mathit{storage\ type\_spec\ declarator\_list}, \sigma\rangle = \langle S', \sigma\rangle,$$

где *storage* есть static при $\mathrm{GLF}_\sigma = 0$, и auto иначе. При этом состояние не изменяется. Но правило перевода **Dec1** из п. 3.2.1 делает то же самое:

$$R(S) = \mathit{storage\ type\_spec\ declarator\_list} = S',$$

т.е. $\mathcal{M}[\![S]\!](\sigma) = \mathcal{M}[\![R(S)]\!](\sigma) = \mathcal{M}[\![S']\!](\sigma)$.

2. Аналогичные рассуждения проводятся для правила **Dec2** из п. 3.2.1 и правила **Dec3** из п. 3.2.2, которые реализуют правила операционной семантики (35) и (36) соответственно.

Таким образом, для любого правила переписывания деклараций R выполнено $S \preccurlyeq_{op} R(S)$, причем отображение на множествах семантических объектов — это просто тождественное отображение.

4.2.2. Корректность правил нормализации

1. Корректность правил **Norm1**–**Norm4**, в которых происходит обрамление тел циклов фигурными скобками и добавление пустой ветви else, доказывается так же, как и в предыдущем пункте. Эти правила в

точности соответствуют правилам операционной семантики (86)–(88) и (73).

2. Правило **Norm5**. Пусть $S \equiv$ `switch(e){A}`, где `A` не содержит `default`. Тогда $R(S) \equiv$ `switch(e){A default: {}}`.

Пусть σ — произвольное состояние. Рассмотрим вычисления, начинающиеся с конфигураций $\langle S, \sigma\rangle$ и $\langle R(S), \sigma\rangle$. Вначале вычисляется значение выражения `e`:

$$\langle \mathtt{e}, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle,$$

причем для обеих программ это вычисление проходит одинаково, поскольку перевод не затрагивает выражение `e`. Поэтому, согласно посылке правил (78) и (78), в обеих программах переходим к конфигурациям

$$\langle \mathtt{A}, \sigma'(\mathrm{GLF}\,{+}{+})\rangle \quad \text{и} \quad \langle \mathtt{A\ default:\ \{\}}, \sigma'(\mathrm{GLF}\,{+}{+})\rangle.$$

Пусть $\sigma'(\mathrm{GLF}\,{+}{+}) = \sigma_1$. Далее вычисляется фрагмент `A`. Заметим, что в семантике частичной корректности бесконечное выполнение (зацикливание) соответствует неопределенному результату. В начале теоремы было сделано замечание о неопределенности, поэтому здесь и далее всегда будем рассматривать только конечные вычисления. Т.е. для некоторого $k \geq 0$

$$\langle \mathtt{A}, \sigma_1\rangle \to_s^k \langle \epsilon, \sigma_2\rangle,$$

а поскольку фрагмент `A` при переводе также не изменяется, получаем конфигурации

$$\langle \epsilon, \sigma_2\rangle \quad \text{и} \quad \langle \mathtt{default:\ \{\}}, \sigma_2\rangle. \qquad (*)$$

Здесь возможны две ситуации:

а) $\mathrm{Val}_{\sigma_2} = \mathrm{BreakVal}$. Тогда левая конфигурация соответствует завершению работы оператора-переключателя, поэтому

$$\langle S, \sigma\rangle \xrightarrow{(79)}_s \langle \epsilon, \sigma_2(\mathrm{GLF}\,{-}{-})(\mathrm{Val} := \mathrm{OkVal})\rangle.$$

Для правой конфигурации получим $\langle \mathtt{default:\ \{\}}, \sigma_2\rangle \xrightarrow{(63)}_s \langle \epsilon, \sigma_2\rangle$. Последняя конфигурация также соответствует завершению работы оператора-переключателя, поэтому

$$\langle R(S), \sigma\rangle \xrightarrow{(79)}_s \langle \epsilon, \sigma_2(\mathrm{GLF}\,{-}{-})(\mathrm{Val} := \mathrm{OkVal})\rangle.$$

Обозначим $\sigma_2(\mathrm{GLF}\,{-}{-})(\mathrm{Val} := \mathrm{OkVal})$ как σ_3. Тогда

$$\mathcal{M}[\![S]\!](\sigma) = \mathcal{M}[\![R(S)]\!](\sigma) = \{\sigma_3\}.$$

б) $\mathrm{Val}_{\sigma_2} = \mathrm{OkVal} \vee \mathrm{CaseVal}(..)$. Как и ранее, в левой конфигурации $(*)$ получается условие выхода из оператора, поэтому

$$\langle S, \sigma\rangle \xrightarrow{(79)}_s \langle \epsilon, \sigma_2(\mathrm{GLF}\,{-}{-})\rangle.$$

Для правой конфигурации получим:

$$\langle \texttt{default: \{\}}, \sigma_2\rangle \xrightarrow{(62)}_s \langle \texttt{\{\}}, \sigma_2\rangle \xrightarrow{(71)}_s \langle \epsilon, \sigma_2\rangle.$$

Поэтому аналогично $\langle R(S), \sigma\rangle \xrightarrow{(79)}_s \langle \epsilon, \sigma_2(\mathrm{GLF}\,{-}{-})\rangle$.

Таким образом, для любого состояния σ $\mathcal{M}[\![S]\!](\sigma) = \mathcal{M}[\![R(S)]\!](\sigma)$, т.е. $S \preccurlyeq_{op} \mathbf{Norm5}(S)$.

Замечание. Схема рассуждений в данном доказательстве далее будет стандартной, кроме тривиальных случаев. Поэтому неформальные пояснения, как правило, будут опускаться.

3. Правило **Norm6**. Пусть $S \equiv$ `if(e) A else B`, где `e` не переменная. Тогда $R(S) \equiv$ `{auto` τ `x; x = e; if(x) A else B}`, где `x` — новая переменная, τ — тип выражения `e`. Пусть σ — произвольное состояние. Начальные конфигурации программ: $\langle S, \sigma\rangle$ и $\langle R(S), \sigma\rangle$.

а) Исполнение первой программы: $\langle \texttt{e}, \sigma\rangle \to_e^* \langle \epsilon, \sigma_a^1\rangle$, где $\mathrm{Val}_{\sigma_a^1} = (v, \tau)$, $IC(\tau, \texttt{int})$ и $v \neq 0$. Тогда

$$\langle \texttt{if(e) A else B}, \sigma\rangle \xrightarrow{(75)}_s \langle \texttt{A}, \sigma_a^1(\mathrm{Val} := \mathrm{OkVal})\rangle.$$

Если далее $\langle \texttt{A}, \sigma_a^1(\mathrm{Val} := \mathrm{OkVal})\rangle \to_s^* \langle \epsilon, \sigma_a^2\rangle$, то $\mathcal{M}[\![S]\!](\sigma) = \{\sigma_a^2\}$.

б) Исполнение второй программы. Вход в блок:

$$\langle \texttt{auto } \tau \texttt{ x; x = e; if(x) A else B}, \sigma_b^0\rangle = \langle S', \sigma_b^0\rangle,$$

где $\sigma_b^0 = \sigma(\mathrm{GLF}\,{+}{+})$. Далее $\langle S', \sigma_b^0\rangle \xrightarrow{(37)}_s \langle \texttt{x = e; if(x) A else B}, \sigma_b^1\rangle$,

где $\sigma_b^1 = \sigma_b^0(\mathrm{ID}^i := \mathrm{ID}^i \cup \{\texttt{x}\})(\mathrm{Adr}^i := \mathrm{Adr}^i \cup \{nc\})(\mathrm{MeM}^i(\texttt{x} \leftarrow nc))$
$(\mathrm{MD}^i(nc \leftarrow 0))(\Gamma^i(\texttt{x} \leftarrow \tau))(\mathrm{Val} := \mathrm{OkVal})$,
$i = \mathrm{GLF}_{\sigma_b^0}$.

Пусть кортеж $Sl = [\mathrm{ID}^i_{\sigma^1_b}, \mathrm{Adr}^i_{\sigma^1_b}, \mathrm{MeM}^i_{\sigma^1_b}, \mathrm{MD}^i_{\sigma^1_b}, \Gamma^i_{\sigma^1_b}, \Sigma^i_{\sigma^1_b}]$. Тем самым $\sigma^1_b = \sigma(\mathrm{GLF}\,{+}{+})(\mathbf{Upd}(Sl))$. Так как $\mathtt{x} \notin \mathtt{e}$, то

$$\langle \mathtt{e}, \sigma(\mathrm{GLF}\,{+}{+})(\mathbf{Upd}(Sl))\rangle \to^*_e \langle \epsilon, \sigma^1_a(\mathrm{GLF}\,{+}{+})(\mathbf{Upd}(Sl))\rangle.$$

Обозначим $\sigma^1_a(\mathrm{GLF}\,{+}{+})(\mathbf{Upd}(Sl))$ как σ_3.

$$\langle \mathtt{x = e; \ if(x) \ A \ else \ B}, \sigma^1_b\rangle \xrightarrow{(22),(55)}_s \langle \mathtt{if(x) \ A \ else \ B}, \sigma_4\rangle,$$

где $\sigma_4 = \sigma_3(\mathrm{MD}^i(\mathrm{MeM}^i(\mathtt{x}) \leftarrow v))(\mathrm{Val} := \mathrm{OkVal})$. Далее

$$\langle \mathtt{if(x) \ A \ else \ B}, \sigma_4\rangle \xrightarrow{(75)}_s \langle \mathtt{A}, \sigma_4\rangle.$$

Т.к. $\sigma_4 = \sigma^1_a(\mathrm{GLF}\,{+}{+})(\mathbf{Upd}(Sl))(\mathrm{MD}^i(\mathrm{MeM}^i(\mathtt{x}) \leftarrow v))(\mathrm{Val} := \mathrm{OkVal})$, а $\mathtt{x} \notin \mathtt{A}$, то

$$\langle \mathtt{A}, \sigma_4\rangle \to^*_s \langle \epsilon, \sigma_5\rangle,$$

где $\sigma_5 = \sigma^2_a(\mathrm{GLF}\,{+}{+})(\mathbf{Upd}(Sl))(\mathrm{MD}^i(\mathrm{MeM}^i(\mathtt{x}) \leftarrow v))$. Наконец срабатывает правило для всего блока:

$$\langle R(S), \sigma\rangle \xrightarrow{(71)}_s \langle \epsilon, \sigma_5(\mathrm{GLF}\,{-}{-})\rangle.$$

По определению операций над флагом $\sigma_5(\mathrm{GLF}\,{-}{-}) \equiv \sigma^2_a$, следовательно, $\mathcal{M}[\![R(S)]\!](\sigma) = \{\sigma^2_a\}$.

Случай ложного значения выражения доказывается аналогично, с той лишь разницей, что вместо фрагмента `A` вычисляется фрагмент `B`. Таким образом $S \preccurlyeq_{op} \mathbf{Norm6}(S)$.

4. Доказательство корректности правила **Norm7** проводится аналогично предыдущему, поскольку оператор-переключатель — это обобщенный условный оператор.

5. Правило **Norm8**. Пусть $S \equiv$ `while(e){A}`, где `A` не содержит `continue`. Тогда $R(S) \equiv$ `while(1){if(e){} else break; A}`. Начальные конфигурации программ: $\langle S, \sigma\rangle$ и $\langle R(S), \sigma\rangle$. По правилу (89) вне зависимости от условного выражения происходит переход к конфигурациям $\langle$`{loop(e,A)}`$, \sigma\rangle$ и $\langle$`{loop(1,if(e){} else break; A)}`$, \sigma\rangle$. Для блока сработает правило (71), поскольку конструкция **loop** сама перехватывает значение GoVal(..), поэтому внешние фигурные скобки можно снять. Согласно семантике частичной корректности цикл — это конструкция с произвольным числом исполнения [6], поэтому покажем, что

$\forall\sigma\forall i \geq 0$ i итераций первого цикла, начинающихся в σ, эквивалентны $i+1$ итерациям второго, начиная с того же состояния. Используем метод индукции.

а) $i = 0$. Такое число итераций для первой программы очевидно соответствует ложному условию. Т.е. $\langle \mathtt{e}, \sigma\rangle \to_e^* \langle \epsilon, \sigma_1\rangle$, где $\mathrm{Val}_{\sigma_1} = (0, \tau)$, $IC(\tau, \mathsf{int})$. Тогда $\langle \mathtt{loop(e,A)}, \sigma\rangle \xrightarrow{(81)}_s \langle \epsilon, \sigma_1(\mathrm{Val} := \mathrm{OkVal})\rangle$. Обозначим вторую программу `loop(1,if(e){} else break; A)` как T. Поскольку условие — константа 1, то

$$\langle \mathtt{loop(1,if(e)\{\} \ else \ break;\ A)}, \sigma\rangle \xrightarrow{(82)}_s \langle T', \sigma\rangle,$$

где $T' \equiv$ `if(e){} else break; A` T. Далее $\langle \mathtt{e}, \sigma\rangle \to_e^* \langle \epsilon, \sigma_1\rangle$, где $\mathrm{Val}_{\sigma_1} = (0, \tau)$, $IC(\tau, \mathsf{int})$. Поэтому

$$\langle \mathtt{if(e)\{\} \ else \ break;\ A}\ T, \sigma\rangle \xrightarrow{(76)}_s \langle \mathtt{break;\ A}\ T, \sigma_1\rangle \xrightarrow{(65)}_s$$
$$\langle \mathtt{A}\ T, \sigma_1(\mathrm{Val} := \mathrm{BreakVal})\rangle \xrightarrow{(69)}_s \langle T, \sigma_1(\mathrm{Val} := \mathrm{BreakVal})\rangle \xrightarrow{(83)}_s$$
$$\langle \epsilon, \sigma_1(\mathrm{Val} := \mathrm{OkVal})\rangle.$$

Т.е. $\langle \mathtt{loop(1,if(e)\{\} \ else \ break;\ A)}, \sigma\rangle \to_s^5 \langle \epsilon, \sigma_1(\mathrm{Val} := \mathrm{OkVal})\rangle$, что совпадает с результатом первой программы и соответствует одному проходу по телу второго цикла.

б) Пусть $i > 0$, в первой программе происходит i итераций, и $\forall j < i$, j итераций первого цикла эквивалентны $j + 1$ итерациям второго при любом входном состоянии. Т.е. значение выражения `e` истинно в ходе всего исполнения. Заметим, что выход из цикла после i повторов может произойти и по причине операторов break и goto, но доказательство для этого случая аналогично.

Для первой программы получим: $\langle \mathtt{e}, \sigma\rangle \to_e^* \langle \epsilon, \sigma_1'\rangle$, где $\mathrm{Val}_{\sigma_1'} = (v, \tau)$, $IC(\tau, \mathsf{int})$ и $v \neq 0$. Тогда

$$\langle \mathtt{loop(e,A)}, \sigma\rangle \xrightarrow{(82)}_s \langle \mathtt{A\ loop(e,A)}, \sigma_1(\mathrm{Val} := \mathrm{OkVal})\rangle.$$

Пусть $\langle \mathtt{A}, \sigma_1(\mathrm{Val} := \mathrm{OkVal})\rangle \to_s^* \langle \epsilon, \sigma_2\rangle$. Тогда остающиеся $i - 1$ итераций начинаются в состоянии σ_2. Вторая программа:

$$\langle \mathtt{loop(1,if(e)\{\} \ else \ break;\ A)}, \sigma\rangle \xrightarrow{(82)}_s \langle T', \sigma\rangle,$$

где T' определено выше. Далее

$$\langle \texttt{if(e)\{\} else break; A } T, \sigma\rangle \xrightarrow{(75)}_s \langle \texttt{\{\} A } T, \sigma_1\rangle \xrightarrow{(71)}_s$$
$$\langle \texttt{A } T, \sigma_1(\mathrm{Val} := \mathrm{BreakVal})\rangle \to_s^* \langle T, \sigma_2\rangle.$$

Т.е. остающиеся итерации во втором цикле тоже начинаются в состоянии σ_2. По предположению индукции во второй программе остаются i эквивалентных итераций. В итоге в первой программе i, а во второй — $i+1$, а заключительные состояния совпадают полностью.

Это означает, что $S \preccurlyeq_{op} \mathbf{Norm8}(S)$.

4.2.3. Корректность правил элиминации операторов for, do-while и switch

1. Правило **LSE1**. $S \equiv$ `switch(x) {A case v: B default: C}`, где `x` — переменная, `A` содержит `case`, `B` не содержит `case`. Тогда $R(S) \equiv$ `switch(x) {A default: if(x == v) {B} C}`. Пусть σ — произвольное состояние. Начальные конфигурации: $\langle S, \sigma\rangle$ и $\langle R(S), \sigma\rangle$. В обеих программах переходим к конфигурациям:

$$\langle \texttt{A case v: B default: C}, \sigma_1\rangle \text{ и } \langle \texttt{A default: if(x == v) \{B\} C}, \sigma_1\rangle,$$

где $\sigma_1 = \sigma(\mathrm{GLF}{+}{+})(\mathrm{Val} := \mathrm{CaseVal}(\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(\texttt{x}))))$. Пусть далее $\langle \texttt{A}, \sigma_1\rangle \to_s^* \langle \epsilon, \sigma_2\rangle$. Возможны три основных ситуации:

а) $\mathrm{Val}_{\sigma_2} = \mathrm{BreakVal}$. Тогда

$$\langle \texttt{case v: B default: C}, \sigma_2\rangle \xrightarrow{(69)}{}^2_s \langle \epsilon, \sigma_2\rangle,$$
$$\langle \texttt{default: if(x == v) \{B\} C}, \sigma_2\rangle \xrightarrow{(69)}_s \langle \epsilon, \sigma_2\rangle.$$

Следовательно, $\langle S, \sigma\rangle \xrightarrow{(79)}_s \langle \epsilon, \sigma_3\rangle$ и $\langle R(S), \sigma\rangle \xrightarrow{(79)}_s \langle \epsilon, \sigma_3\rangle$, где $\sigma_3 = \sigma_2(\mathrm{GLF}{-}{-})(\mathrm{Val} := \mathrm{OkVal})$.

б) $\mathrm{Val}_{\sigma_2} \neq \mathrm{BreakVal} \vee \mathrm{OkVal}$ и $\mathrm{MD}_{\sigma_1}(\mathrm{MeM}(\texttt{x})) \neq \texttt{v}$. Тогда в первой программе:

$$\langle \texttt{case v: B default: C}, \sigma_2\rangle \xrightarrow{(61)}_s \langle \texttt{default: C}, \sigma_2\rangle \xrightarrow{(62)}_s$$
$$\langle \texttt{C}, \sigma_2(\mathrm{Val} := \mathrm{OkVal})\rangle = \langle \texttt{C}, \sigma_3\rangle.$$

Пусть $\langle \texttt{C}, \sigma_3\rangle \to_s^* \langle \epsilon, \sigma_4\rangle$, тогда $\langle S, \sigma\rangle \xrightarrow{(78)}_s \langle \epsilon, \sigma_4(\mathrm{GLF}{-}{-})\rangle$

Во второй программе:

$$\langle \texttt{default: if(x == v) \{B\} C}, \sigma_2\rangle \xrightarrow{(62)}_s$$
$$\langle \texttt{if(x == v) \{B\} C}, \sigma_2(\mathrm{Val} := \mathrm{OkVal})\rangle \xrightarrow{\texttt{x}\notin\texttt{A}\ \text{и}\ (76)}_s$$
$$\langle \texttt{C}, \sigma_2(\mathrm{Val} := \mathrm{OkVal})\rangle = \langle \texttt{C}, \sigma_3\rangle.$$

Следовательно, $\langle \texttt{C}, \sigma_3\rangle \to_s^* \langle \epsilon, \sigma_4\rangle$ и $\langle R(S), \sigma\rangle \xrightarrow{(78)}_s \langle \epsilon, \sigma_4(\mathrm{GLF}--)\rangle$.

в) $\mathrm{Val}_{\sigma_2} = \mathrm{OkVal} \vee \mathrm{CaseVal}(\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(\texttt{x})))$ и $\mathrm{MD}_{\sigma_1}(\mathrm{MeM}(\texttt{x})) = \texttt{v}$. В этом случае доказательство проводится аналогично, в обеих программах дополнительно будет исполняться фрагмент `B` в одинаковых контекстах.

Т.е. во всех случаях получаются эквивалентные программы. Это означает, что $S \preccurlyeq_{op} \mathbf{LSE1}(S)$.

2. Корректность правила **LSE2** доказывается аналогично, поскольку при пустом фрагменте `A` — это частный случай правила **LSE1**.

3. Правило **LSE3**. $S \equiv$ `switch(x) {default: A case v: B}`, где `x` — переменная, `A` не содержит `case` на верхнем уровне. Тогда $R(S) \equiv$ `switch(x){default: {if(x == v) goto l; A l: B}}`. При доказательстве корректности правила **LSE1** было показано, что запись `if(x == v) B` эквивалентна `case v: B`. Новый оператор `goto l;` передаст управление в обход фрагмента `A`, но это не изменяет семантики. По условию `A` не содержит `case` на верхнем уровне, следовательно, в исходной программе он будет просто игнорироваться по правилу семантики (69).

Это означает, что $S \preccurlyeq_{op} \mathbf{LSE3}(S)$.

4. Правило **LSE4**. $S \equiv$ `switch(x) {default: A}`, где `x` — переменная, `A` не содержит `case` и `break` на верхнем уровне. Тогда $R(S) \equiv$ `{A}`. Пусть σ — произвольное состояние. Для первой программы получим:

$$\langle \texttt{default: A}, \sigma(\mathrm{GLF}++)(\mathrm{Val} := \mathrm{CaseVal}(\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(\texttt{x}))))\rangle \xrightarrow{(62)}_s$$
$$\langle \texttt{A}, \sigma(\mathrm{GLF}++)(\mathrm{Val} := \mathrm{OkVal})\rangle \to_s^* \langle \epsilon, \sigma_1\rangle.$$

Тогда $\langle S, \sigma\rangle \xrightarrow{(78)}_s \langle \epsilon, \sigma_1(\mathrm{GLF}--)\rangle$.

Для второй программы получим:

$$\langle \texttt{A}, \sigma(\mathrm{GLF}++)(\mathrm{Val} := \mathrm{OkVal})\rangle \to_s^* \langle \epsilon, \sigma_1\rangle,$$

отсюда $\langle R(S), \sigma\rangle \xrightarrow{(78)}_s \langle \epsilon, \sigma_1(\mathrm{GLF}--)\rangle$.

Необходимо отметить важность условия, что `A` не содержит `case` и `break` на верхнем уровне. Без него удаление фрагментов `switch(x)` и `default:` привело бы к синтаксической ошибке.

Таким образом $S \preccurlyeq_{op} \mathbf{LSE4}(S)$.

5. Правило **LSE5**. Пусть $S \equiv$ `do {A} while(e)`, где `A` не содержит `continue`. Тогда $R(S) \equiv$ `while(1) {A if (e) {} else break;}`. Эта программа получается из

$$R' \equiv \texttt{while(1) \{if (e) \{\} else break; A\}}$$

перестановкой фрагмента `A`. Эта перестановка соответствует тому, как в цикле do..while тело выполняется до проверки условия. В правиле семантики (90) происходит такой же вынос тела, после которого начинает работать цикл **loop**, который в точности есть цикл while. Эквивалентность программы R' циклу while была доказана в п. 4.2.2. Отсюда $S \preccurlyeq_{op} \mathbf{LSE5}(S)$.

6. Правило **LSE6**. Пусть $S \equiv$ `for(e1; e2; e3) {A}`, где `A` не содержит `continue`. Тогда $R(S) \equiv$ `{e1; while(e2) {A e3}}`. Пусть σ — произвольное состояние.

Для первой программы получим переход:

$$\langle \texttt{for(e1; e2; e3) \{A\}}, \sigma\rangle \xrightarrow{(91)}_s$$
$$\langle \texttt{e1; if(!e2) break; A } \mathbf{loop}\texttt{((e3,e2),A)}, \sigma\rangle.$$

Таким образом первым оператором, как и в R(S), становится `e1;`. Пусть $\langle$`e1;`$, \sigma\rangle \to_s \langle \epsilon, \sigma_1\rangle$. Тогда для программ получим конфигурации:

$$\langle \texttt{if(!e2) break; A } \mathbf{loop}\texttt{((e3,e2),A)}, \sigma_1\rangle \quad \text{и} \quad \langle \mathbf{loop}\texttt{(e2, A e3)}, \sigma_1\rangle.$$

Как и ранее, эквивалентность двух циклов можно доказать по индукции, однако очевидно следующее: а) при ложном значении выражения `e2` в первой программе сработает break, что позволит проигнорировать первую же итерацию цикла, а во второй программе итерации не произойдет по определению **loop**; б) при истинном `e2` оба цикла начнут работу, причем всякий раз после `A` и до начала нового вычисления `e2` будет вычисляться выражение `e3`. Более того, после первой итерации получатся совершенно эквивалентные программы:

$$\langle \mathbf{loop}\texttt{((e3,e2),A)}, \sigma_2\rangle \quad \text{и} \quad \langle \mathbf{loop}\texttt{(e2, A e3)}, \sigma_1\rangle$$

для некоторого σ_2. Важно отметить, что эти программы эквивалентны только при условии правила **LSE6**: `A` не содержит `continue`. Именно это ограничение позволяет выносить выражение `e3` в условие цикла.

Это означает, что $S \preccurlyeq_{op} \mathbf{LSE6}(S)$.

4.2.4. Корректность правил элиминации операторов continue и break

1. Правило **BCE1**. $S \equiv$ `switch(e) {A break; B}`. Тогда $R(S) \equiv$ `{switch(e){A goto l; B} l:}`, где `l` — новая метка. Пусть σ — произвольное состояние.

Первая программа. Пусть $\langle \mathtt{e}, \sigma\rangle \to_e^* \langle \epsilon, \sigma_1\rangle$. Тогда вход в тело оператора switch приводит к

$$\langle \mathtt{A\ break;\ B}, \sigma_1(\mathrm{GLF}\ {+}{+})\rangle \to_s^* \langle \mathtt{break;\ B}, \sigma_2\rangle \xrightarrow{(65)}_s$$
$$\langle \mathtt{B}, \sigma_2(\mathrm{Val} := \mathrm{BreakVal})\rangle \xrightarrow{(69)}_s^* \langle \epsilon, \sigma_2(\mathrm{Val} := \mathrm{BreakVal})\rangle.$$

Тогда $\langle S, \sigma\rangle \xrightarrow{(79)}_s \langle \epsilon, \sigma_2(\mathrm{GLF}\ {-}{-})(\mathrm{Val} := \mathrm{OkVal})\rangle$.

Вторая программа. Обрамление скобками не влияет на выражение `e`: $\langle \mathtt{e}, \sigma(\mathrm{GLF}\ {+}{+})\rangle \to_e^* \langle \epsilon, \sigma_1(\mathrm{GLF}\ {+}{+})\rangle$. Тогда

$$\langle \mathtt{A\ goto\ l;\ B}, \sigma_1(\mathrm{GLF}\ {+}{+})(\mathrm{GLF}\ {+}{+})\rangle \to_s^*$$
$$\langle \mathtt{goto\ l;\ B}, \sigma_2(\mathrm{GLF}\ {+}{+})\rangle \xrightarrow{(64)}_s$$
$$\langle \mathtt{B}, \sigma_2(\mathrm{GLF}\ {+}{+})(\mathrm{Val} := \mathrm{GoVal}(\mathtt{l}))\rangle \xrightarrow{\mathtt{l}\notin \mathtt{B}, (69)}_s^*$$
$$\langle \epsilon, \sigma_2(\mathrm{GLF}\ {+}{+})(\mathrm{Val} := \mathrm{GoVal}(\mathtt{l}))\rangle.$$

Тогда

$$\langle \mathtt{switch(e)\{A\ goto\ l;\ B\}\ l:}, \sigma(\mathrm{GLF}\ {+}{+})\rangle \xrightarrow{(78)}_s$$
$$\langle \mathtt{l:}, \sigma_2(\mathrm{GLF}\ {+}{+})(\mathrm{Val} := \mathrm{GoVal}(\mathtt{l}))(\mathrm{GLF}\ {-}{-})\rangle \xrightarrow{(57)}_s$$
$$\langle \epsilon, \sigma_2(\mathrm{Val} := \mathrm{OkVal})\rangle.$$

Следовательно, $\langle R(S), \sigma\rangle \xrightarrow{(72)}_s \langle \epsilon, \sigma_2(\mathrm{GLF}\ {-}{-})(\mathrm{Val} := \mathrm{OkVal})\rangle$.

Это означает, что $\mathcal{M}[\![S]\!](\sigma) = \mathcal{M}[\![R(S)]\!](\sigma)$, т.е. $S \preccurlyeq_{op} \mathbf{BCE1}(S)$.

2. Правило **BCE2**. Пусть $S \equiv$ `while(e){A break; B}`. Тогда $R(S) \equiv$ `{while(e){A goto l; B} l:}`, где `l` — новая метка. Пусть σ — про-

извольное состояние. Очевидно, что в случае ложного значения условия цикла программы эквивалентны, поскольку оба цикла будут просто пропущены. Возможны две ситуации:

а) первая программа:

$$\langle \texttt{while(e) \{A break; B\}}, \sigma\rangle \xrightarrow{(89)}_s \langle \{\textbf{loop}(\texttt{e}, \texttt{A break; B})\}, \sigma\rangle.$$

Пусть $\langle \texttt{e}, \sigma(\text{GLF} + +)\rangle \to_e^* \langle \epsilon, \sigma_1\rangle$, где $\text{Val}_{\sigma_1} = (v, \tau)$, $IC(\tau, \mathsf{int})$ и $v \neq 0$. Тогда

$$\langle \textbf{loop}(\texttt{e}, \texttt{A break; B}), \sigma_1\rangle \xrightarrow{(82)}_s \langle \texttt{A break; B } \textbf{loop}(\texttt{e}, \texttt{A break; B}), \sigma_2\rangle,$$

где $\sigma_2 = \sigma_1(\text{Val} := \text{OkVal})$. Пусть $\langle \texttt{A}, \sigma_2\rangle \to_s^* \langle \epsilon, \sigma_3\rangle$ и $\text{Val}_{\sigma_3} = \text{OkVal}$. Тогда

$$\begin{aligned}
&\langle \texttt{break; B } \textbf{loop}(\texttt{e}, \texttt{A break; B}), \sigma_3\rangle \xrightarrow{(65)}_s \\
&\langle \texttt{B } \textbf{loop}(\texttt{e}, \texttt{A break; B}), \sigma_3(\text{Val} := \text{BreakVal})\rangle \xrightarrow{(69),(83)}_s^* \\
&\langle \epsilon, \sigma_3(\text{Val} := \text{OkVal})\rangle = \langle \epsilon, \sigma_4\rangle.
\end{aligned}$$

Тогда $\langle S, \sigma\rangle \xrightarrow{(71)}_s \langle \epsilon, \sigma_4(\text{GLF} - -)\rangle$.

Вторая программа:

$$\begin{aligned}
&\langle \texttt{while(e) \{A goto l; B\}}, \sigma(\text{GLF} + +)\rangle \xrightarrow{(89)}_s \\
&\quad \langle \{\textbf{loop}(\texttt{e}, \texttt{A goto l; B})\}, \sigma(\text{GLF} + +)\rangle.
\end{aligned}$$

Тогда, как и выше,

$$\begin{aligned}
&\langle \textbf{loop}(\texttt{e}, \texttt{A goto l; B}), \sigma_1(\text{GLF} + +)\rangle \xrightarrow{(82)}_s \\
&\langle \texttt{A goto l; B } \textbf{loop}(\texttt{e}, \texttt{A goto l; B}), \sigma_2(\text{GLF} + +)\rangle \to_s^* \\
&\langle \texttt{goto l; B } \textbf{loop}(\texttt{e}, \texttt{A goto l; B}), \sigma_3(\text{GLF} + +)\rangle \xrightarrow{(64)}_s \\
&\langle \texttt{B } \textbf{loop}(\texttt{e}, \texttt{A goto l; B}), \sigma_3(\text{GLF} + +)(\text{Val} := \text{BreakVal})\rangle \xrightarrow{(69),(83)}_s^* \\
&\langle \epsilon, \sigma_3(\text{GLF} + +)(\text{Val} := \text{OkVal})\rangle = \langle \epsilon, \sigma_4(\text{GLF} + +)\rangle.
\end{aligned}$$

Тогда $\langle R(S), \sigma\rangle \xrightarrow{(71)}{}_s^2 \langle \epsilon, \sigma_4(\text{GLF} - -)\rangle$.

б) вторая ситуация соответствует тому, что после $\langle \mathtt{A}, \sigma_2\rangle \to_s^* \langle \epsilon, \sigma_3\rangle$ $(\mathrm{Val}_{\sigma_3} = \mathrm{ContVal}) \vee (\mathrm{Val}_{\sigma_3} = \mathrm{GoVal(L)} \wedge \mathrm{L} \in \mathtt{B})$. В этом случае не произойдет принудительного выхода из цикла и начнутся новые итерации. Как и в случае правила **Norm8**, это соответствует индуктивному переходу, и корректность доказывается аналогично.

Таким образом $S \preccurlyeq_{op} \mathbf{BCE2}(S)$.

3. Правила **BCE3**. Пусть $S \equiv$ `while(e) {A continue; B}`. Тогда $R(S) \equiv$ `while(e) {A goto l; B l:}`, где `l` — новая метка. Очевидно корректность этого правила доказывается аналогично предыдущему, причем этот случай проще. Управление по goto передается в пределах тела цикла, поэтому нет дополнительных фигурных скобок вокруг нового цикла.

4. Корректность правил **BCE3** и **BCE4** доказывается аналогично предыдущему случаю, поскольку по семантике все циклы автоматически преобразуются к конструкции **loop**, которая в точности соответствует циклу while.

4.2.5. Корректность правил элиминации логических операций

1. Правило **Logic1**. $S \equiv \mathtt{e||e'}$. Тогда $R(S) \equiv$ `(e?1:e')`. Пусть σ — произвольное состояние. Начальные конфигурации программ: $\langle S, \sigma\rangle$ и $\langle R(S), \sigma\rangle$. Возможны два случая:

а) пусть $\langle \mathtt{e}, \sigma\rangle \to_e^* \langle \epsilon, \sigma_1\rangle$, где $\mathrm{Val}_{\sigma_1} = (v, \tau)$, τ — скалярный и $v \neq 0$. По правилам (14) и (19), соответственно, переходим к конфигурациям

$$\langle \epsilon, \sigma_1(\mathrm{Val} := (\mathsf{TRUE}, \mathsf{bool}))\rangle \quad \text{и} \quad \langle (\mathsf{bool})1, \sigma_1\rangle.$$

Для второй конфигурации очевиден переход

$$\langle (\mathsf{bool})1, \sigma_1\rangle \xrightarrow{(11)}_e \langle \epsilon, \sigma_1(\mathrm{Val} := (\mathsf{TRUE}, \mathsf{bool}))\rangle,$$

т.е. заключительные состояния совпадают полностью.

б) пусть $\langle \mathtt{e}, \sigma\rangle \to_e^* \langle \epsilon, \sigma_1\rangle$, где $\mathrm{Val}_{\sigma_1} = (0, \tau)$, τ — скалярный. По правилам (16) и (20), соответственно, переходим к конфигурациям

$$\langle \mathbf{OrAnd}(\mathtt{e'}), \sigma_1\rangle \quad \text{и} \quad \langle (\mathsf{bool})\mathtt{e'}, \sigma_1\rangle.$$

Если $\langle \mathtt{e'}, \sigma_1\rangle \to_e^* \langle \epsilon, \sigma_2\rangle$, где $\mathrm{Val}_{\sigma_2} = (0, \tau')$ и τ' — скалярный, то для обеих программ получаем конфигурацию $\langle \epsilon, \sigma_2(\mathrm{Val} := (\mathsf{FALSE}, \mathsf{bool}))\rangle$, иначе $\langle \epsilon, \sigma_2(\mathrm{Val} := (\mathsf{TRUE}, \mathsf{bool}))\rangle$.

Таким образом, $\forall\sigma\ \mathcal{M}[\![S]\!](\sigma) = \mathcal{M}[\![R(S)]\!](\sigma)$, т.е. $S \preccurlyeq_{op} \mathbf{Logic1}(S)$, причем отображение на множествах семантических объектов тождественно.

2. Для правила **Logic2** рассуждения аналогичны, поскольку операция И двойственна операции ИЛИ, поэтому также будет возникать форма **OrAnd**, но с другими истинностными значениями.

Для правила **Logic3** рассуждения еще проще, так как это одноместная операция, не содержащая контрольную точку. Более того, переписывание происходит в точности с определением этой операции: ноль — это логическая ложь, единица — логическая истина.

4.2.6. Корректность правила элиминации операции выборки из массива

Рассмотрим фрагмент $S \equiv \texttt{a[e]}$. Пусть σ — произвольное состояние. Пусть $\Gamma_\sigma(\texttt{a}) = \mathrm{lv}[\tau[n]]$ для некоторого n. Вначале вычисляем индекс: $\langle \texttt{e}, \sigma\rangle \to_e^* \langle \epsilon, \sigma_1\rangle$, где $\mathrm{Val}_{\sigma_1} = (v, \tau')$, τ' — скалярный. Вычисляем адрес: $\langle \texttt{a}, \sigma_1\rangle \to_e^* \langle \epsilon, \sigma_2\rangle$, где $\mathrm{Val}_{\sigma_2} = (c, \tau*)$. Тогда

$$\langle \texttt{a[i]}, \sigma\rangle \xrightarrow{(5)}_e \langle \epsilon, \sigma_3\rangle,$$

где $\sigma_3 = \sigma_2(\mathrm{Val} := (\mathrm{MD}_{\sigma_2}(c + v), \tau))$.

Рассмотрим фрагмент $R(S) \equiv \texttt{*(a+i)}$. Семантика операции сложения реализована в абстрактной функции BinOpSem, которая по типам аргументов определит, что получается адресное выражение. Адресное и индексное выражения при переводе не изменяются, поэтому

$$\langle \texttt{a+i}, \sigma\rangle \xrightarrow{(9)}_e \langle \epsilon, \sigma_2(\mathrm{Val} := (\mathrm{BinOpSem}(+, c, \tau*, v, \tau'), \tau*))\rangle.$$

Очевидно, что в данном случае $\mathrm{BinOpSem}(+, c, \tau*, v, \tau') = c + v$. Тогда

$$\langle \texttt{*(a+i)}, \sigma\rangle \xrightarrow{(7)}_e \langle \epsilon, \sigma_2(\mathrm{Val} := (\mathrm{MD}_{\sigma_2}(c + v), \tau))\rangle = \langle \epsilon, \sigma_3\rangle.$$

Таким образом, $\forall\sigma\ \mathcal{M}[\![S]\!](\sigma) = \mathcal{M}[\![R(S)]\!](\sigma)$, т.е. $S \preccurlyeq_{op} \mathbf{Ind}(S)$, причем отображение на множествах семантических объектов тождественно.

4.2.7. Корректность правил элиминации операций инкремента и декремента

1. Корректность правила **Inc1** очевидно следует из того, что оно реализует правило семантики (23).

2. Правило **Inc2**. Рассмотрим исполнение фрагмент $S \equiv$ `e++`. Тогда $R(S) \equiv$ `(q = &e, y = *q, *q = *q + 1, y)`, где переменные `q` и `y` имеют типы выражений `&e` и `e` соответственно. Пусть σ — произвольное состояние. Начальные конфигурации: $\langle S, \sigma\rangle$ и $\langle R(S), \sigma\rangle$.

Вначале вычисляем `e`: $\langle \mathtt{e}, \sigma\rangle \to_e^* \langle \epsilon, \sigma_1\rangle$, где $\mathrm{Val}_{\sigma_1} = (v, \tau)$ и пусть $\mathrm{MeM}_\sigma(\mathtt{e}) = c$. Тогда

$$\langle \mathtt{e{+}{+}}, \sigma\rangle \xrightarrow{(24)}_e \langle \epsilon, \sigma_1(\mathrm{MD}(c \leftarrow v+1))\rangle$$

$$\langle R(S), \sigma\rangle \xrightarrow{(8),(22)}_e \langle \mathtt{y = *q,\ *q = *q + 1,\ y}, \sigma_2\rangle,$$

где $\sigma_2 = \sigma_1(\mathrm{MD}(\mathrm{MeM}(\mathtt{q}) \leftarrow c))$. Далее

$$\langle \mathtt{y = *q,\ *q = *q + 1,\ y}, \sigma_2\rangle \xrightarrow{(7),(22),(12)}_e$$

$$\langle \mathtt{*q = *q + 1,\ y}, \sigma_2(\mathrm{MD}(\mathrm{MeM}(\mathtt{y}) \leftarrow v))(\mathrm{Val} := \mathrm{OkVal})\rangle \xrightarrow{(7),(22),(12)}_e$$

$$\langle \mathtt{y}, \sigma_2(\mathrm{MD}(\mathrm{MeM}(\mathtt{y}) \leftarrow v))(\mathrm{MD}(c \leftarrow v+1))(\mathrm{Val} := \mathrm{OkVal})\rangle \xrightarrow{(3)}_e$$

$$\langle \epsilon, \sigma_2(\mathrm{MD}(\mathrm{MeM}(\mathtt{y}) \leftarrow v))(\mathrm{MD}(c \leftarrow v+1))(\mathrm{Val} := (v, \tau))\rangle.$$

По определению $\sigma_2 = \sigma_1(\mathrm{MD}(\mathrm{MeM}(\mathtt{q}) \leftarrow c))$ и $\mathrm{Val}_{\sigma_1} = (v, \tau)$. Это означает, что $S \preccurlyeq_{op} \mathbf{Inc2}(S)$, причем отображение на множествах семантических объектов не тождественная биекция, а взаимно-однозначное вложение.

4.2.8. Корректность правила элиминации составных присваиваний

Рассмотрим это правило на примере операции `+=`. Пусть фрагмент $S \equiv$ `e += e'`. Тогда $R(S) \equiv$ `(x = e', y = &e, *y = *y + x)`. Переменные `x` и `y` имеют типы выражений `e'` и `&e` соответственно. σ — произвольное состояние, в котором `e` : lv[τ], `e'` : τ', а также выполнено $IC(\tau, \tau')$.

Первая программа — $\langle S, \sigma\rangle$. Вычисляются выражения:

$$\langle \mathtt{e'}, \sigma\rangle \to_e^* \langle \epsilon, \sigma_1\rangle \quad \text{и} \quad \langle \mathtt{e}, \sigma_1\rangle \to_e^* \langle \epsilon, \sigma_2\rangle,$$

где $\mathrm{Val}_{\sigma_1} = (v', \tau')$ и $\mathrm{Val}_{\sigma_2} = (v, \tau)$. Пусть $\mathrm{MeM}_{\sigma_1}(\mathtt{e}) = c$. Тогда

$$\langle \mathtt{e \ += \ e'}, \sigma \rangle \xrightarrow{(21)}_e \langle \epsilon, \sigma_3 \rangle,$$

где $\sigma_3 = \sigma_2(\mathrm{MD}(c \leftarrow res))(\mathrm{Val} := (res, \tau))$
$res = \mathrm{BinOpSem}(+, v', \tau', v, \tau)$.

Вторая программа — $\langle R(S), \sigma \rangle$. Пусть $\mathrm{MeM}(\mathtt{x}) = c_x$, а $\mathrm{MeM}(\mathtt{y}) = c_y$. Выражения не изменились, а новые переменные в них не входят, поэтому

$\langle R(S), \sigma \rangle \xrightarrow{(22),(12)}_e$

$\langle \mathtt{y = \&e, \ *y = *y + x}, \sigma_1(\mathrm{MD}(c_x \leftarrow v'))(\mathrm{Val} := \mathrm{OkVal}) \rangle \xrightarrow{(8),(22),(12)}_e$

$\langle \mathtt{*y = *y + x}, \sigma_2(\mathrm{MD}(c_x \leftarrow v')(c_y \leftarrow c))(\mathrm{Val} := \mathrm{OkVal}) \rangle \xrightarrow{(7),(22),(12)}_e$

$\langle \epsilon, \sigma_3' \rangle$,

$\sigma_3' = \sigma_2(\mathrm{MD}(c_x \leftarrow v')(c_y \leftarrow c))(\mathrm{MD}(\mathrm{MD}(c_y) \leftarrow res'))(\mathrm{Val} := (res', \tau))$, а $res' = \mathrm{BinOpSem}(+, v', \tau', \mathrm{MD}(c), \tau)$.

Но $\mathrm{MD}(c) = v$, а $\mathrm{MD}(c_y) = c$. Т.е. σ_3' отличается от σ_3 только наличием информации о новых переменных. Следовательно, $S \preccurlyeq_{op} \mathbf{As}(S)$.

4.2.9. Корректность правила декомпозиции выражений

Докажем корректность правила для случаев вызова функции и операции сложения, что не ограничивает общности.

а) пусть фрагмент $S \equiv \mathtt{f(e_1, \ ..., \ e_{i-1}, \ e_i, \ e_{i+1}, \ ..., \ e_n)}$, где $\mathtt{f}$ — некоторая функция, определенная в программе, а σ — произвольное состояние, в котором $\mathtt{f} : \tau_1 \times ... \times \tau_n \to \tau$. По условию $\mathtt{e_{i+1}}, ..., \mathtt{e_n}$ являются либо переменными, либо константами, поэтому вычисляются за один шаг каждое:

$$\langle \mathtt{e_n}, \sigma \rangle \to_e \langle \epsilon, \sigma^n_{arg} \rangle, \quad \ldots, \quad \langle \mathtt{e_{i+1}}, \sigma^{i+2}_{arg} \rangle \to_e \langle \epsilon, \sigma^{i+1}_{arg} \rangle.$$

Здесь $\mathrm{Val}_{\sigma^n_{arg}} = (v_n, \tau'_n), ..., \mathrm{Val}_{\sigma^{i+1}_{arg}} = (v_{i+1}, \tau'_{i+1})$ и $\forall j = i+1, ..., n$ выполнено $IC(\tau_j, \tau'_j)$. Далее вычисляются остальные аргументы:

$$\langle \mathtt{e_i}, \sigma^{i+1}_{arg} \rangle \to^*_e \langle \epsilon, \sigma^i_{arg} \rangle, \quad \ldots, \quad \langle \mathtt{e_1}, \sigma^2_{arg} \rangle \to^*_e \langle \epsilon, \sigma^1_{arg} \rangle.$$

Здесь $\mathrm{Val}_{\sigma^i_{arg}} = (v_i, \tau'_i), ..., \mathrm{Val}_{\sigma^1_{arg}} = (v_1, \tau'_1)$ и $\forall j = 1, ..., i$ $IC(\tau_j, \tau'_j)$. Таким образом

$$\langle \mathtt{f(e_1, ..., e_i, ..., e_n)}, \sigma \rangle \xrightarrow{(29)}_e \langle \mathbf{FCall}(\mathtt{f})(v_1, ..., v_i, ..., v_n), \sigma^1_{arg} \rangle.$$

Пусть *parms* — список деклараций параметров функции `f`, а B — ее тело, и пусть $\langle B, InstParms(\sigma^1_{arg}(\text{GLF} ++),\ parms, [v_1, ..., v_n])\rangle \to^*_s \langle \epsilon, \sigma_{end}\rangle$, где $\text{Val}_{\sigma_{end}} = \text{RetVal}(val, \tau')$ и $IC(\tau, \tau')$. Тогда

$$\langle S, \sigma\rangle \xrightarrow{(30)}_e \langle \epsilon, \sigma_{end}(\text{Val} := (val, \tau))(\text{GLF} --)\rangle = \langle \epsilon, \sigma'\rangle.$$

Рассмотрим фрагмент

$$R(S) \equiv \texttt{(x = e}_\texttt{i}\texttt{, f(e}_\texttt{1}\texttt{, ..., e}_{\texttt{i}-\texttt{1}}\texttt{, x, e}_{\texttt{i}+\texttt{1}}\texttt{, ..., e}_\texttt{n}\texttt{))}.$$

Переменная $\texttt{x} \notin \texttt{e}_\texttt{i}$, поэтому

$$\langle \texttt{e}_\texttt{i}, \sigma\rangle \to^*_e \langle \epsilon, \sigma_0\rangle \quad \Longrightarrow \quad \langle R(S), \sigma\rangle \xrightarrow{(22)}_e \langle S', \sigma_0(\text{MD}(\text{MeM}(\texttt{x}) \leftarrow v_i))\rangle,$$

где $S' = \texttt{f(e}_\texttt{1}\texttt{, ..., e}_{\texttt{i}-\texttt{1}}\texttt{, x, e}_{\texttt{i}+\texttt{1}}\texttt{, ..., e}_\texttt{n}\texttt{)}$, $\text{Val}_{\sigma_0} = (v_i, \tau'_i)$. Обозначим $\sigma_0(\text{MD}(\text{MeM}(\texttt{x}) \leftarrow v_i))$ как θ. Аргумент в $i-$й позиции теперь тоже переменная, поэтому

$$\langle \texttt{e}_\texttt{n}, \theta\rangle \to_e \langle \epsilon, \theta^n_{arg}\rangle, \quad \ldots, \quad \langle \texttt{e}_\texttt{i}, \theta^{i+1}_{arg}\rangle \to_e \langle \epsilon, \theta^i_{arg}\rangle.$$

Здесь $\text{Val}_{\theta^n_{arg}} = (v_n, \tau'_n), ..., \text{Val}_{\theta^i_{arg}} = (v_i, \tau'_i)$ и $\forall j = i, ..., n\ IC(\tau_j, \tau'_j)$. Вычисляем остальные аргументы-выражения, которые не изменились, т.е

$$\langle \texttt{e}_{\texttt{i}-\texttt{1}}, \theta^i_{arg}\rangle \to^*_e \langle \epsilon, \theta^{i-1}_{arg}\rangle, \quad \ldots, \quad \langle \texttt{e}_\texttt{1}, \theta^2_{arg}\rangle \to^*_e \langle \epsilon, \theta^1_{arg}\rangle.$$

Здесь $\text{Val}_{\theta^{i-1}_{arg}} = (v_{i-1}, \tau'_{i-1}), ..., \text{Val}_{\theta^1_{arg}} = (v_1, \tau'_1)$ и $\forall j = 1, ..., i-1$ выполнено $IC(\tau_j, \tau'_j)$. Далее

$$\langle \texttt{f(e}_\texttt{1}\texttt{, ..., x, ..., e}_\texttt{n}\texttt{)}, \sigma\rangle \xrightarrow{(29)}_e \langle \textbf{FCall}(\texttt{f})(v_1, ..., v_i, ..., v_n), \sigma^1_{arg}\rangle.$$

Вынос вычисления выражения $\texttt{e}_\texttt{i}$ вперед не влияет на вычисление более правых аргументов, поскольку все они либо переменные, либо константы. И обратно, вычисление аргумента, являющегося переменной или константой, не влияет на вычисление остальных аргументов. Переменная `x` — новая и не входит ни в одно из выражений, поэтому

$$\forall i\ \theta^i_{arg} = \sigma^i_{arg}(\text{MD}(\text{MeM}(\texttt{x}) \leftarrow v_i)).$$

Следовательно,

$$\langle B, InstParms(\theta^1_{arg}(\text{GLF} ++), parms, [v_1, ..., v_n])\rangle \to^*_s \langle \epsilon, \theta_{end}\rangle,$$

и $\theta_{end} = \sigma_{end}(\mathrm{MD}(\mathrm{MeM}(\mathtt{x}) \leftarrow v_i))$. Отсюда

$$\langle R(S), \sigma\rangle \to_e^2 \langle \epsilon, \sigma'(\mathrm{MD}(\mathrm{MeM}(\mathtt{x}) \leftarrow v_i))\rangle.$$

b) пусть $S \equiv \mathtt{e_1} + \mathtt{e_2}$. Возможны две ситуации: 1) $\mathtt{e_2}$ — не переменная и не константа; 2) $\mathtt{e_2}$ — переменная или константа, но $\mathtt{e_1}$ — сложное выражение. Рассмотрим доказательство для первой ситуации, как для более сложной. Тогда $R(S) \equiv$ `(x = e2, e1 + x)`. Пусть σ — произвольное состояние. Начальные конфигурации программ: $\langle S, \sigma\rangle$ и $\langle R(S), \sigma\rangle$.

Первая программа. Пусть

$$\langle \mathtt{e_2}, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle \quad \text{и} \quad \langle \mathtt{e_1}, \sigma'\rangle \to_e^l \langle \epsilon, \sigma''\rangle,$$

где $\mathrm{Val}_{\sigma'} = (v, \tau)$ и $\mathrm{Val}_{\sigma''} = (u, \tau')$. Тогда

$$\langle \mathtt{e_1} + \mathtt{e_2}, \sigma\rangle \xrightarrow{(9)}_e \langle \epsilon, \sigma'''\rangle,$$

где $\sigma''' = \sigma''(\mathrm{Val} := \mathrm{BinOpSem}(+, v, \tau, u, \tau'))$.

Вторая программа:

$$\langle \mathtt{x = e_2,\ e_2 + x}, \sigma\rangle \xrightarrow{(22),(12)}_e$$
$$\langle \mathtt{e_2 + x}, \sigma'(\mathrm{MD}(\mathrm{MeM}(\mathtt{x}) \leftarrow v))\rangle \xrightarrow{\mathtt{x} \notin \mathtt{e_1},(9)}_e \langle \epsilon, \sigma'''(\mathrm{MD}(\mathrm{MeM}(\mathtt{x}) \leftarrow v))\rangle.$$

Таким образом, для обоих случаев $S \preccurlyeq_{op} \mathbf{Ops}(S)$, причем отображения семантических объектов — взаимно-однозначные вложения. Число статических объектов не изменяется.

4.2.10. Корректность правила нормализации левой части операции присваивания

Доказательство корректности этого правила совершенно аналогично доказательству для правила **CAs** в п. 4.2.8, поскольку правило **LHS** отличается только тем, что происходит простое присваивание без применения бинарной операции. Это означает, что $S \preccurlyeq_{op} \mathbf{LHS}(S)$.

4.2.11. Корректность правил нормализации операции взятия адреса

Эти два правила вообще не изменяют состояний с точки зрения компонент, поскольку таковы операции адресации и косвенной адресации,

а выражение `e` остается без изменения. Поэтому покажем, что значения выражений совпадают.

Пусть $S \equiv$ `*&e` и σ — произвольное состояние. Рассмотрим все возможные варианты выражения `e`, которое в данном случае должно быть модифицируемым L-значением. В языке C-light запрещены взятия адреса для выражений выбора элемента и выражения приведения типа, поэтому остаются следующие варианты:

a) `e` — идентификатор простой переменной. Тогда

$$\langle \texttt{*\&e}, \sigma \rangle \xrightarrow{(7),(8)}_e \langle \epsilon, \sigma(\mathrm{Val} := (\mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(\texttt{e})), \Gamma(\texttt{e})))\rangle.$$

Но это и есть значение выражения-переменной `e` по определению состояния абстрактной машины.

b) `e` — это индексное выражение вида `a[i]`, где `a` — массив типа `int[n]`. Пусть $\mathrm{MeM}(\texttt{a}) = c_1$ и $\mathrm{MeM}(\texttt{i}) = c_2$, а $c_1 + \mathrm{MD}(c_2) = c_3$. Поскольку `a[i]` $\equiv$ `*(a + i)`, то значение всего выражения `*&a[i]` есть $\mathrm{Val}_\sigma = \mathrm{MD}_\sigma(\mathrm{MeM}_\sigma(\mathrm{MD}_\sigma(c_3))) = \mathrm{MD}(c_3)$, т.е. значение выражения `e`.

c) `e` — это выражение косвенной адресации вида `*x`, где `x` указатель типа `int*`. Пусть $\mathrm{MeM}(\texttt{x}) = c_1$, $\mathrm{MD}(c_1) = loc$, где константа $loc \in \mathrm{Adr}_\sigma$, а $\mathrm{MD}(loc) = v$ для некоторого целочисленного значения v, т.е. это значение выражения `e`. Можно взять адрес L-выражения: $\mathrm{MeM}_\sigma(\texttt{*x}) = loc$. Таким образом, значение исходного выражения `*&*x` есть $\mathrm{MD}(loc) = v$.

d) `e` — это L-выражение в скобках. Доказательство очевидно.

Это означает, что $S \preccurlyeq_{op} \mathbf{Adr1}(S)$, причем отображение семантических объектов — тождественная биекция. Новые статические объекты не появляются.

Для выражения $S \equiv$ `&*e` ситуация симметрична. В этом случае `e` может быть только простым указателем либо идентификатором массива. Поэтому в доказательстве просто меняются местами компоненты определения адреса и значения в состояниях.

4.2.12. Корректность правила элиминации условной операции

Пусть $S \equiv$ `e1?e2:e3;` и σ — произвольное состояние.

a) пусть $\sigma \models \texttt{e1} : \tau$ и

$$\langle \texttt{e1}, \sigma \rangle \to_e^* \langle \epsilon, \sigma' \rangle,$$

где $\mathrm{Val}_{\sigma'} = (v, \tau)$ и $v \neq 0$. В этом случае выполнится `e2`. Пусть

$$\langle \texttt{e2}, \sigma' \rangle \to_e^* \langle \epsilon, \sigma'' \rangle,$$

где $\mathrm{Val}_{\sigma'} = (w, \tau_2)$. Тогда

$$\langle S, \sigma\rangle \xrightarrow{(55)}_s \langle \epsilon, \sigma'''\rangle,$$

где $\sigma''' = \sigma''(\mathrm{Val} := \mathrm{OkVal})$.

Рассмотрим $R(S) \equiv$ `{auto x = e1; if(x) {e2;} else {e3;}}`. Раскрывая скобки, получим

$$\langle R(S), \sigma\rangle \to_s \langle \texttt{auto x = e1; if(x) \{e2;\} else \{e3;\}}, \sigma_1\rangle = \langle T, \sigma_1\rangle,$$

где $\sigma_1 = \sigma(\mathrm{GLF}\,{+}{+})$. Выражение `e1` не изменилось, поэтому

$$\langle \texttt{e1}, \sigma_1\rangle \to_e^* \langle \epsilon, \sigma'(\mathrm{GLF}\,{+}{+})\rangle = \langle \epsilon, \sigma_2\rangle,$$

где $\mathrm{Val}_{\sigma_2} = (v, \tau)$ и $v \neq 0$. Отсюда

$$\langle T, \sigma_1\rangle \to_s \langle \texttt{if(x) \{e2;\} else \{e3;\}}, \sigma_3\rangle,$$

где $\sigma_3 = \sigma_2(\mathrm{ID}^i := \mathrm{ID}^i \cup \{\texttt{x}\})(\mathrm{Adr}^i := \mathrm{Adr}^i \cup \{NewConst\})$
$(\mathrm{MeM}^i(\texttt{x} \leftarrow NewConst))(\mathrm{MD}^i(NewConst \leftarrow v))$
$(\Gamma^i(\texttt{x} \leftarrow \tau))(\mathrm{Val} := \mathrm{OkVal})$,
$i = \mathrm{GLF}_{\sigma_2}$.

Тогда

$$\langle \texttt{if(x)\{e2;\} else \{e3;\}}, \sigma_3\rangle \to_s \langle \texttt{\{e2;\}}, \sigma_3\rangle \to_s \langle \texttt{e2;\}}, \sigma_4\rangle,$$

где $\sigma_4 = \sigma_3(\mathrm{GLF}\,{+}{+})$. Очевидно $\mathrm{GLF}_{\sigma_4} = \mathrm{GLF}_{\sigma'} + 2$ и σ_4 отличается от σ' наличием множеств с номерами $\mathrm{GLF}_{\sigma'} + 1$ и $\mathrm{GLF}_{\sigma'} + 2$. Переменная $\texttt{x} \notin \texttt{e2}$, поэтому

$$\langle \texttt{e2;\}}, \sigma_4\rangle \to_s \langle \texttt{\}}, \sigma_5\rangle,$$

где $\sigma_5 = \sigma'''(\mathrm{GLF}\,{+}{+})(\mathrm{ID}^i := \mathrm{ID}^i \cup \{\texttt{x}\})(\mathrm{Adr}^i := \mathrm{Adr}^i \cup \{NewConst\})$
$(\mathrm{MeM}^i(\texttt{x} \leftarrow NewConst))(\mathrm{MD}^i(NewConst \leftarrow v))$
$(\Gamma^i(\texttt{x} \leftarrow \tau))(\mathrm{GLF}\,{+}{+})$,
$i = \mathrm{GLF}_{\sigma'''}$.

Тогда правило для блока приведет к уничтожению разницы между этими состояниями. Это означает, что $\mathcal{M}[\![S]\!](\sigma) = \mathcal{M}[\![T]\!](\sigma) = \{\sigma'''\}$.

b) в случае, когда значение `e1` есть ноль, будут вычисляться ветви else условного выражения и условного оператора. Доказательство проводится аналогично.

Это означает, что $S \preccurlyeq_{op} \mathbf{Tern}(S)$.

4.2.13. Корректность правил нормализации правой части операции присваивания

1. Правило **RHS1**. Рассмотрим $S \equiv \texttt{x = e?e}_1\texttt{:e}_2$. Пусть σ — произвольное состояние.

а) пусть $\sigma \models \texttt{e} : \tau$ для некоторого скалярного типа τ и

$$\langle \texttt{e}, \sigma\rangle \to_e^* \langle \epsilon, \sigma'\rangle,$$

где $\mathrm{Val}_{\sigma'} = (v, \tau)$ и $v \neq 0$. Пусть в σ также выполнено $\texttt{e}_1 : \tau_1$, $\texttt{e}_2 : \tau_2$ и для некоторого типа τ_{Res} истинен предикат $P_?(\tau_1, \tau_2, \tau_{Res})$ из п. 2.3. Тогда

$$\langle \texttt{e}_1, \sigma'\rangle \to_e^* \langle \epsilon, \sigma_1\rangle,$$

где $\mathrm{Val}_{\sigma_1} = (w, \tau_{Res})$ для некоторого w. Инициализация переменной:

$$\langle \texttt{x = e?e}_1\texttt{:e}_2, \sigma_0\rangle \xrightarrow{(22)}_e \langle \epsilon, \sigma_1(\mathrm{MD}(\mathrm{MeM}(\texttt{x}) \leftarrow \Gamma_\sigma(\texttt{x})(w)))\rangle = \langle \epsilon, \sigma_{Res}\rangle.$$

Рассмотрим фрагмент $R(S) \equiv \texttt{e?(x = e}_1\texttt{):(x = e}_2\texttt{)}$. Выражение e не изменилось, поэтому

$$\langle \texttt{e?(x = e}_1\texttt{):(x = e}_2\texttt{)}, \sigma\rangle \xrightarrow{(19)}_e \langle \texttt{(x = e}_1\texttt{)}, \sigma'\rangle.$$

Тогда

$$\langle \texttt{(x = e}_1\texttt{)}, \sigma'\rangle \xrightarrow{(22)}_e \langle \texttt{x = e}_1, \sigma'\rangle \to_e \langle \epsilon, \sigma_1(\mathrm{MD}(\mathrm{MeM}(\texttt{x}) \leftarrow \Gamma_\sigma(\texttt{x})(w)))\rangle,$$

т. е. $\langle R(S), \sigma\rangle \to_e^3 \langle \epsilon, \sigma_{Res}\rangle$.

б) как и в случае логических операций, если выражение e ложно, то вычисляется выражение $\texttt{e}_2$ и доказательство проводится аналогично.

Это означает, что $S \preccurlyeq_{op} \mathbf{RHS1}(S)$, причем отображение семантических объектов — тождественная биекция.

2. Правило **RHS2**. Рассмотрим $S \equiv \texttt{x = e}_1\texttt{, . . . , e}_\texttt{n}$. Пусть σ_0 — произвольное состояние.

а) вначале последовательно вычисляется правая часть, т. е. $\forall\, i.\ 1 \leq i \leq n\ \exists\, l_i \geq 0$:

$$\langle \texttt{e}_\texttt{i}, \sigma_{i-1}\rangle \to_e^{l_i} \langle \epsilon, \sigma_i\rangle,$$

и $\mathrm{Val}_{\sigma_n} = (v, \tau)$. Инициализация соответствующей области памяти:

$$\langle \texttt{x = e}_1\texttt{, . . . , e}_\texttt{n}, \sigma_0\rangle \xrightarrow{(22)}_e \langle \epsilon, \sigma_n(\mathrm{MD}(\mathrm{MeM}(\texttt{x}) \leftarrow \Gamma_{\sigma_0}(\texttt{x})(v)))\rangle = \langle \epsilon, \sigma'\rangle.$$

Рассмотрим $R(S) \equiv$ `(e1, ..., x = en)`. Изменилось только последнее подвыражение в цепочке. Пусть $k = \sum_{i=1}^{n-1} l_i$. Тогда

$$\langle R(S), \sigma_0 \rangle \to_e^k \langle \texttt{x = en}, \sigma_{n-1} \rangle.$$

Вычисление левой части присваивания приводит к σ_n. Таким образом,

$$\langle R(S), \sigma_0 \rangle \to_e^{k+1} \langle \epsilon, \sigma_n(\mathrm{MD}(\mathrm{MeM}(\texttt{x}) \leftarrow \Gamma_{\sigma_0}(\texttt{x})(v)))\rangle = \langle \epsilon, \sigma' \rangle.$$

Это означает, что $S \preccurlyeq_{op} \mathbf{RHS2}(S)$, причем отображение семантических объектов — тождественная биекция.

4.2.14. Корректность правила элиминации операции «запятая»

Рассмотрим $S \equiv$ `e1,...,en;`. Пусть σ_0 — произвольное состояние. Операнды вычисляются слева-направо, и значением всего выражения будет значение последнего, т.е. $\forall\, i.\ 1 \le i \le n\ \exists\, l_i \ge 0$:

$$\langle \texttt{ei}, \sigma_{i-1} \rangle \to_e^{l_i} \langle \epsilon, \sigma_i \rangle.$$

Тогда

$$\langle S, \sigma_0 \rangle \xrightarrow{(55)}_s \langle \epsilon, \sigma_n \rangle.$$

Рассмотрим $R(S) \equiv$ `{e1; ...; en;}`. Тогда

$$\langle T, \sigma_0 \rangle \to_s \langle \texttt{e1;...;en;}\}, \sigma_0(\mathrm{GLF}{+}{+}) \rangle = \langle \texttt{e1;...;en;}\}, \sigma_0' \rangle.$$

Выражения `ei` не изменились, поэтому

$$\langle \texttt{ei}, \sigma_{i-1}' \rangle \to_e^{l_i} \langle \epsilon, \sigma_i' \rangle \quad \text{и} \quad \langle \texttt{ei;}, \sigma_{i-1}' \rangle \xrightarrow{(55)}_s \langle \epsilon, \sigma_i' \rangle,$$

где $\sigma_i' = \sigma_i(\mathrm{GLF}{+}{+})$. Таким образом,

$$\langle R(S), \sigma_0 \rangle \xrightarrow{(71)}_s \langle \epsilon, \sigma_n \rangle.$$

В исходных выражениях не могли содержаться декларации, поэтому добавление фигурных скобок не сужает области определенности каких-либо объектов. Следовательно, $\mathcal{M}[\![S]\!](\sigma_0) = \mathcal{M}[\![R(S)]\!](\sigma_0) = \{\sigma_n\}$.

Это означает, что $S \preccurlyeq_{op} \mathbf{Comma}(S)$, причем отображение семантических объектов — тождественная биекция.

4.2.15. Корректность правила нормализации скобок

Пусть $S \equiv$ `(e)`, а σ — произвольное состояние. Пусть для некоторого $k \geq 0$ $\langle \texttt{(e)}, \sigma \rangle \to_e^k \langle \epsilon, \sigma' \rangle$. По семантике выражений при рекурсивном спуске по выражению до операндов скобки удаляются. При этом значение всего выражения в круглых скобках — это значение выражения, обрамленного скобками.

Рассмотрим $R(S) \equiv$ `e`. Условие применимости правила в точности означает, что выражение вычисляется как единое целое, без разбиения на части, которые являются операндами других бинарных операций. В вызове функции такое переписывание может быть некорректным, так как может произойти увеличение числа фактических аргументов. Таким образом, без скобок цепочка переходов по отношению для вычисления выражений будет на единицу короче:

$$\langle \texttt{e}, \sigma \rangle \to_e^{k-1} \langle \epsilon, \sigma' \rangle.$$

Это означает, что $S \preccurlyeq_{op} \mathbf{Pars}(S)$.

■

Следствие 1. Правила перевода из языка C-light в C-light-kernel сохраняют свойство частичной корректности, т.е. для любой тройки Хоара $\langle P \rangle S \langle Q \rangle$ и любого правила R, если $\models \langle P \rangle S \langle Q \rangle$, то $\models R(\langle P \rangle S \langle Q \rangle)$.

Доказательство. По доказанной теореме ни одно правило перевода не изменяет семантику программы относительно исходных объектов. Также ни одно правило не изменяет спецификации программы. Т.е. пред- и постусловия, а также инварианты циклов остаются утверждениями над исходными объектами программы. Тем самым, если $\mathcal{M}[\![S]\!](\|P\|) \subseteq \|Q\|$, то $\mathcal{M}[\![R(S)]\!](\|P\|) \subseteq \|Q\|$, что по определению [6] есть условие истинности тройки Хоара в смысле частичной корректности. ■

Следствие 2. Для любой программы S на языке C-light и для любого правила перевода R из языка C-light в C-light-kernel $S \simeq_{op} R(S)$.

Доказательство. В доказательстве **Теоремы 1** было показано, что ни одно правило не увеличивает число статических объектов. Также невозможно отождествить статический и автоматический объекты при отображении ϕ, поскольку атрибут объекта включает информацию о его классе памяти. Таким образом, в качестве биекций из определения $\simeq_{op}$ достаточно брать сужения на статические объекты тех отображений, которые строятся в доказательстве. ■

4.3. Теорема о завершимости перевода

Теорема 2. Для любой синтаксически правильной программы на языке C-light процесс перевода завершается.

Доказательство. Пусть Z_+ — множество неотрицательных целых чисел. В качестве фундированного порядка на нем положим обычный порядок «больше» $>$. Порядок на кортежах определим как лексикографическое расширение порядка на элементах кортежа, а порядок на мультимножествах — как расширение порядка на элементах мультимножества на мультимножества. Расширение фундированного порядка на мультимножества и лексикографическое расширение фундированных порядков сохраняет фундированность. Поэтому любое множество, построенное с помощью конструкторов кортежей и мультимножеств, и в качестве базового множества которого берется Z_+, является фундированным. Доказательство завершимости правил преобразования состоит в определении для каждой группы правил функции $termin$, отображающей программу в фундированное множество, построенное с помощью выше описанных средств, и в установлении факта, что для любой программы P выполнено свойство $termin(P) > termin(P')$, где P' — результат применения правила преобразования из соответствующей группы.

1. Докажем завершимость правил для деклараций. Опишем функцию $termin$, преобразующую программу в фундированное множество пар $W = Z_+ \times Z_+$. Функция $termin$, получая на вход программу P, выдает пару (n, m), где n — число деклараций в программе P, содержащих более одного объявления переменных, m — число деклараций в P, которые не имеют спецификатора auto или static.

Правило **Dec3** уменьшает первый элемент пары $termin(P)$. Правила **Dec1** и **Dec2** уменьшают второй элемент пары $termin(P)$ и не изменяют ее первый элемент. Поэтому для любой программы P имеем $termin(P) \succ termin(P')$, где P' — результат применения правил для деклараций к P. Тогда, в силу фундированности W, правила для деклараций завершаются.

2. Докажем завершимость правил для операторов. Опишем функцию $termin$, преобразующую программу в фундированное множество пар W. Первый элемент пары — мультимножество пар неотрицательных целых чисел, а второй — неотрицательное целое число. Функция

$termin$, получая на вход программу P, выдает пару (N, n), где n — сумма следующих слагаемых:

a) число операторов switch, do-while, for, break и continue в P;
b) число тел циклов do-while, while и for в P, не являющихся блоками;
c) число операторов if в P без ветви else;
d) число операторов switch в P без ветви default;
e) число условий операторов if и switch в P, отличных от переменной;
f) число условий оператора while в P, отличных от константы 1.

N определяется следующим образом: 1) для каждого фрагмента вида A case B, входящего в P, вычисляется число операторов switch, попадающих в область действия данного case, 2) мультимножество всех таких неотрицательных целых чисел образует N.

Правила **LSE1** – **LSE3** уменьшают первый элемент пары $termin(P)$. Правила **Norm1** – **Norm8**, **LSE4** – **LSE6**, **BCE1** – **BCE5** уменьшают второй элемент пары $termin(P)$ и, по крайней мере, не увеличивают ее первый элемент. Поэтому для любой программы P имеем $termin(P) \succ termin(P')$, где P' — результат применения правила для операторов к P. Тогда, в силу фундированности W, правила для операторов завершаются.

3. Докажем завершимость правил для выражений. Опишем функцию $termin$, преобразующую программу в фундированное множество W одинадцатиэлементных кортежей. Все элементы кортежа — неотрицательные целые числа, на которых определен порядок $>$. Функция $termin$, получая на вход программу P, выдает кортеж элементов n_i, где $1 \leq i \leq 11$:

n_1 — число логических операций в P;
n_2 — число операций выборки из массива в P;
n_3 — число операций декремента и инкремента в P;
n_4 — число составных присваиваний в P;
n_5 — число выражений, отличных от переменной и являющихся аргументами функций или операндами операций, входящих в P;
n_6 — число левых частей операции присваиваний в P вида, отличного от z или $*z$, где z — переменная;
n_7 — число операций взятия адреса в P;
n_8 — число условных операций в P;
n_9 — число правых частей операции присваивания в P вида, начинающихся с условной операции или операции запятая;
n_{10} — число операций запятая в P;

n_{11} — число круглых скобок в P.

Правила для выражений структурированы так, что применение i-й группы правил к программе P приводит к уменьшению i-го элемента кортежа $termin(P)$ и, по крайней мере, не увеличивает элементы этого кортежа с меньшими номерами. Поэтому для любой программы P имеем $termin(P) \succ termin(P')$, где P' — результат применения правила для выражений к P. Тогда, в силу фундированности W, правила для выражений завершаются.

Таким образом, система правил перевода завершается.

■

4.4. Теорема о нормальной форме

Теорема 3. Для любой синтаксически правильной программы на языке C-light программа, получаемая в результате перевода, является программой на языке C-light-kernel.

Доказательство. 1. На множество деклараций языка C-light-kernel накладываются следующие ограничения:

a) все переменные имеют явные спецификаторы auto или static;

b) переменные не имеют других спецификаторов, кроме спецификаторов auto и static;

c) каждая декларация описывает не более одной переменной.

Первое ограничение обеспечивается правилами **Dec1** и **Dec2**, второе — правилом **Dec2**, третье — правилом **Dec3**, преобразующим декларацию, содержащую несколько объявлений переменных, в последовательность деклараций, объявляющих только по одной переменной.

2. На множество операторов C-light-kernel по сравнению с операторами C-light накладываются следующие три ограничения:

a) оператор if всегда имеет ветвь else;

b) условия операторов if и while — выражения без побочных эффектов;

c) операторов for, do-while, switch, continue и break нет.

Первые два ограничения обеспечиваются группой правил нормализации, а третье — группами правил элиминации операторов. Группа правил нормализации также гарантирует выполнение требования, что условие оператора if — переменная, а условие оператора while — константа 1. Это требование дает даже более сильное ограничение, чем второе ограничение языка C-light-kernel для операторов.

3. Оператор выражения C-light-kernel имеет вид: 1) $e = e'$ или 2) e', где, в свою очередь, e имеет вид x или $*x$ для некоторой переменной x, а e' имеет вид x *op* y, c, $\&x$ или $f(x_1, ..., x_n)$ для некоторых переменных x, x_1, ..., x_n и y, константы c, операции

$$op \in \{+,\ -,\ *,\ /,\ <,\ >,\ <=,\ =>,\ !=,\ ==\}$$

и функции f. Условие на выражение e обеспечивается правилом нормализации левой части операции присваивания, а условие на выражение e' — правилами нормализации правой части операции присваивания и операции взятия адреса, а также правилом декомпозиции. Ограничение на операцию *op* обеспечивается правилами элиминации операций.

Таким образом, результирующая программа является программой на языке C-light-kernel.

■

5. ЗАКЛЮЧЕНИЕ

Описанная в настоящей работе формальная система правил перевода из языка C-light в язык C-light-kernel играет важную роль в двухуровневой схеме верификации C-light программ. Обоснование корректности этой системы правил не имеет аналогов в литературе и требует глубоких семантических исследований.

Оказалось, что обычное отношение функциональной эквивалентности между исходной C-light программой и результирующей C-light-kernel программой неприменимо, и его пришлось ослабить посредством введения отношения семантического расширения, которое сохраняет свойства рефлексивности и транзитивности, однако не сохраняет свойства симметричности. В результате удалось провести полное доказательство корректности системы правил перевода, которое включает также доказательства завершимости процесса перевода и нормализации результирующего представления. Это позволило обосновать важное свойство, что частичная корректность исходной аннотированной C-light программы следует из частичной корректности результирующей C-light-kernel программы.

На основе данной системы правил перевода был создан прототип транслятора из языка C-light в язык C-light-kernel. Предполагается разработать и реализовать экспериментальную систему верификации C-light программ, которая включает следующие основные части:

- упомянутый транслятор в язык C-light-kernel;
- генератор условий корректности, который базируется на аксиоматической семантике языка C-light-kernel;
- доказатель условий корректности, включающий базы знаний о проблемных областях.

Описание этой системы верификации и экспериментов будет материалом четвертой части работы по верификации C-light программ.

СПИСОК ЛИТЕРАТУРЫ

1. **Керниган Б., Ритчи Д.** Язык программирования Си. — М.: Финансы и статистика, 1985.
2. **ISO/IEC 9899:1990**, Programming languages — C, 1990.
3. **ISO/IEC 9899:1999**, Programming languages — C, 1999.
4. **Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.** На пути к верификации С-программ. Язык C-light. — Материалы конф., посвященной 90-летию со дня рождения А.А. Ляпунова. — Новосибирск, 2001. — С. 423-432.
5. **Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.** На пути к верификации С программ. Часть 1. Язык C-light. — Новосибирск, 2001. — 48 с. — (Препр. / РАН. Сиб. Отд-ние. ИСИ; № 84).
6. **Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.** На пути к верификации С программ. Часть 2. Язык C-light-kernel и его аксиоматическая семантика. — Новосибирск, 2001. — 58 с. — (Препр. / РАН. Сиб. Отд-ние. ИСИ; № 87).
7. **Непомнящий В.А., Рякин О.М.** Прикладные методы верификации программ. — М.: Радио и связь, 1988.
8. **Apt K.R., Olderog E.R.** Verification of sequential and concurrent programs. — Berlin a.o.: Springer Verlag, 1991.
9. **Black P.E., Windley Ph.J.** Inference rules for programming languages with side effects in expressions // Proc. 9th Intern. Conf. on Theorem Proving in HOL. — Berlin a.o.: Springer Verlag, 1996. — P. 56–60. — (Lect. Notes Comput. Sci.; Vol. 1125).
10. **Elgaard J., Moller A., Schwartzbach M.I.** Compile-time debugging of C programs working on trees // Proc. Europ. Symp. on Programming (ESOP2000). — Berlin a.o.: Springer Verlag, 2000. — P. 119–134. — (Lect. Notes Comput. Sci.; Vol. 1782).
11. **Fradet P., Gaugne R., Le Metayer D.** Static detection of pointer errors: an axiomatisation and a checking algorithm // Proc. Europ. Symp. on Programming (ESOP96). — Berlin a.o.: Springer Verlag, 1996. — P. 125–140. — (Lect. Notes Comput. Sci.; Vol. 1058).
12. **Gurevich Y., Huggins J.K.** The semantics of the C programming language // Proc. of the Intern. Conf. on Computer Science Logic. — Berlin a.o.: Springer Verlag, 1993. — P. 274–309. — (Lect. Notes Comput. Sci.; Vol. 702).
13. **Huggins J.K., Shen W.** The static and dynamic semantics of C (extended abstract) // Local Proc. Intern. Workshop on Abstract State Machines. — Zurich,

2000. — P. 272–284. — (ETH TIK-Rep.; N 87).
14. **Luckham D., Suzuki N.** Verification of array, record and pointer operations in Pascal // ACM Trans. on Programming Languages and Systems, 1979. — Vol. 1, N 2. — P. 226–244.
15. **Norrish M.** Deterministic expressions in C // Proc. Europ. Symp. on Programming (ESOP99). — Berlin a.o.: Springer Verlag, 1999. — P. 147–161. — (Lect. Notes Comput. Sci.; Vol. 1576).
16. **Norrish M.** C formalised in HOL: PhD thesis. — Computer Lab., Univ. of Cambridge, 1998.
17. **Plotkin G.D.** A structure approach to operational semantics. — 1981. —(Tech. Rep., Aarhus University, DAIMI; N FN-19).

В. А. Непомнящий, И. С. Ануреев,
И. Н. Михайлов, А. В. Промский

НА ПУТИ К ВЕРИФИКАЦИИ С-ПРОГРАММ. ЧАСТЬ 3. ПЕРЕВОД ИЗ ЯЗЫКА C-LIGHT В ЯЗЫК C-LIGHT-KERNEL И ЕГО ФОРМАЛЬНОЕ ОБОСНОВАНИЕ

Препринт
97

Рукопись поступила в редакцию 28.03.2002
Рецензент Ф. А. Мурзин
Редактор З. В. Скок

Подписано в печать 14.06.2002
Формат бумаги 60×84 1/16 Объем 4,7 уч.-изд.л., 5,1 п.л.
Тираж 50 экз.

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