Модел. и анализ информ. систем. Т. **17**, № **3** (2010) 5–28

УДК 519.681.3

Верификация С-программ на основе смешанной аксиоматической семантики

Ануреев И. С., Марьясов И. В., Непомнящий В. А.

Институт систем информатики им. А. П. Ершова СО РАН
Новосибирский государственный университет

e-mail: anureev@iis.nsk.su, ivm1999@mail.ru, vnep@iis.nsk.su

получена 25 мая 2010

Ключевые слова: верификация, операционная семантика, аксиоматическая семантика, язык C, язык C-light, язык C-kernel, частичная корректность

Описывается смешанная аксиоматическая семантика языка C-kernel, являющегося ядром представительного подмножества языка C, названного C-light. Такая семантика позволяет во многих случаях упростить условия корректности верифицируемых программ. Данная семантика является основой разрабатываемого генератора условий корректности C-kernel-программ. Рассмотрен пример, иллюстрирующий применение правил вывода описанной семантики.

1. Введение

Формальная верификация программ — актуальное направление современного программирования. Особый интерес представляет верификация программ, написанных на распространённых языках системного программирования, таких как C и C++. Обозримая формальная семантика является необходимой предпосылкой того, что язык удобен для верификации.

Формальной детерминированной семантики для полного языка C, соответствующего стандарту ISO C [9], не существует. В лаборатории теоретического программирования ИСИ СО РАН был разработан язык C-light, являющийся представительным подмножеством языка C. Формально этот язык был определён с помощью структурной операционной семантики в стиле Плоткина. Хотя операционная семантика удобна для формального определения языка, для верификации она имеет слишком низкий уровень, что приводит к громоздким доказательствам условий корректности. Поэтому обычно используют аксиоматическую семантику, базирующуюся на логике Хоара [8], которая определяется как система вывода утверждений о свойствах программ. Однако аксиоматическая семантика для языка C-light также была бы громоздкой. В связи с этим был применён двухуровневый подход: в языке

C-light выделяется ядро — язык C-kernel, для которого разработана аксиоматическая семантика [1, 4], и в этот язык транслируются исходные программы на языке C-light. По сравнению с языком C-light в языке C-kernel более простые выражения и более ограниченный набор операторов, что упростило аксиоматическую семантику. В [1] была доказана важная теорема о непротиворечивости аксиоматической семантики языка C-kernel относительно операционной, а также были описаны формальные правила перевода из языка C-light в язык C-kernel и дано доказательство их корректности.

Верификация реальных программ (особенно среднего и большого объёма) — процесс трудоёмкий, поэтому его целесообразно автоматизировать. Однако предложенная в [1, 4] семантика не ориентирована на автоматизацию процесса вывода условий корректности, поскольку использует унифицированную модель памяти языка C, где для каждой программной переменной вводится понятие адреса и значения, находящегося по данному адресу. Это приводило к громоздким условиям корректности даже для программ без указателей, что затрудняло их последующее доказательство.

Цель данной работы — описать смешанную аксиоматическую семантику. Термин "смешанная" означает, что в неё введены правила вывода специального вида, применяемые, когда в анализируемом фрагменте программы нет переменных, доступ к значению которых осуществляется через указатели. При этом данная семантика обладает однозначностью вывода (т. е. на каждом шаге анализа в ней допустимо одно и только одно правило вывода). Это позволяет во многих случаях существенно упростить условия корректности. Предварительная версия данной семантики была представлена на международном семинаре [10].

Работа имеет следующую структуру. Раздел 2 даёт обзор языков C-light и C-kernel, в разделе 3 вводится язык утверждений, используемый для написания спецификаций. Раздел 4 посвящён операционной семантике языка C-light, в разделе 5 приведены правила вывода смешанной аксиоматической семантики языка C-kernel. Раздел 6 посвящён проблеме перевода инвариантов циклов при трансляции программ из языка C-light в язык C-kernel. Пример, иллюстрирующий применение смешанной аксиоматической семантики, приведён в разделе 7.

Работа частично поддержана грантом РФФИ 08-01-00899-а.

2. Языки C-light и C-kernel

Язык C-light является представительным подмножеством языка C99 [9], расширенным операциями выделения и освобождения памяти языка C++. Формальную грамматику C-light можно найти в [5]. Рассмотрим подробнее ограничения этого языка.

В языке C-light есть пустой, целочисленные, вещественные типы, перечисления. Из составных типов присутствуют указатели, массивы, структуры и функции. Комплексные типы и объединения не допускаются. Имеются некоторые дополнительные ограничения, среди которых запрет неполных типов массивов (кроме случая использования в аргументах функций), битовых полей и функций с переменным числом аргументов.

Неокончательные определения запрещены. Не допускаются абстрактные декларации аргументов. Использование спецификатора *static* в размере массива, являющегося параметром функции, запрещено. Не допускается использование любых спецификаторов и модификаторов типов, кроме класса памяти, наличия знака и размера для скалярных типов. Имена всех статических объектов в программе уникальны.

В языке C-light строго фиксирован порядок вычисления выражений: аргументы операций и функций вычисляются справа налево, списки инициализирующих выражений слева направо.

Составные литералы в C-light разрешены только в качестве инициализаторов в декларациях. Операция приведения типа для составного инициализатора запрещена.

Приведения типов указателей ограничены случаем приведения от типа $void*$ к произвольному τ.

Для работы с динамической памятью используются операции *new* и *delete* из языка C++. Это позволяет унифицировать и формализовать работу с динамической памятью, не обращаясь к разнородным и неформализованным функциям работы с памятью стандартной библиотеки языка C.

На операторы накладывается два ограничения: все *case*-метки и метка $default$ в операторе *switch* должны находиться на одном уровне вложенности; запрещён переход по оператору *goto* внутрь блока.

Весь исходный текст программы есть последовательность внешних деклараций. На внешнем уровне можно объявить тип ($typedef$-декларации), функцию (прототип или определение), структуру и данные.

В языке C-light препроцессор отсутствует, поскольку исходная программа препроцессируется до начала её верификации. Модульность не поддерживается, поэтому с точки зрения семантики любая исходная программа состоит из одного файла.

Язык C-kernel — это промежуточный язык двухуровневой схемы верификации программ, в который транслируются программы на языке C-light.

Число побочных эффектов в выражениях языка C-kernel сведено до минимума, а операции, содержащие контрольные точки (например, логические && и ||), отсутствуют. Нормализованным выражением называется выражение, не содержащее условных операций, операций последовательного вычисления, логических операций && и ||, простых и составных присваиваний, операций инкремента и декремента. Пусть e (возможно, с индексами) обозначает нормализованное выражение, τ — его тип. Оператор-выражение языка C-kernel имеет вид:

- $e = e'(e_1, \ldots, e_n)$;
- $e = new(\tau)$;
- $e = e'$;
- $e'(e_1, \ldots, e_n)$;
- $delete(e)$.

Списки декларируемых объектов разрешены только в декларациях функций, любая другая декларация объявляет ровно один объект. Декларации находятся в нормальной форме и содержат в инициализирующей части только нормализованные выражения. Спецификаторы класса памяти *static* и *auto* обязательны, других спецификаторов класса памяти не допускается.

В языке C-kernel допустимыми являются следующие операторы:

1. Оператор вычисления выражения (возможно, пустой).
2. Условный оператор if с обязательной ветвью $else$ (возможно, пустой) и нормализованным условием.
3. Оператор цикла $while$ с нормализованным условием.
4. Оператор передачи управления $goto$ с теми же ограничениями, что и в C-light.
5. Оператор возврата значения $return$, причём возвращаемым выражением может быть только нормализованное выражение.
6. Составной оператор (блок).

Правила трансляции программ на языке C-light в программы на языке C-kernel, а также формальное доказательство свойств корректности и завершимости процесса перевода даны в [1, 2].

3. Язык утверждений

Для описания свойств абстрактной машины, определяющей операционную семантику языка C-light, и для записи аннотаций в аксиоматической семантике языка C-kernel используется единый язык утверждений. Главным требованием к этому языку является точное описание всех объектов программы и взаимосвязей между ними.

Типы. Типами выражений языка утверждений являются следующие типы:

1. $CTypes$ — объединение всех допустимых типов языка C-light.
2. $Names$ — множество имён, встречающихся в программе.
3. $Locations$ — множество адресов в памяти.
4. $TypeSpecs$ — множество абстрактных имён типов.
5. $LogTypeSpecs$ — множество логических имён типов, которые являются логическими представлениями абстрактных имён типов. Оно определяется следующим образом:
   - $\tau \in LogTypeSpecs$, если τ — базовый тип;
   - $pointer(\tau) \in LogTypeSpecs$, если $\tau \in LogTypeSpecs$;
   - $\tau_1 \times \tau_2 \times \ldots \times \tau_n \mapsto \tau \in LogTypeSpecs$, если $\tau_1, \ldots, \tau_n, \tau \in LogTypeSpecs$;
   - $struct(s, (\tau_1, m_1), \ldots, (\tau_n, m_n)) \in LogTypeSpecs$, если $\tau_1, \ldots, \tau_n \in LogTypeSpecs$ и $s, m_1, \ldots, m_n, \tau \in Names$;
   - $enum(s, m_1, \ldots, m_n) \in LogTypeSpecs$, если $s, m_1, \ldots, m_n \in Names$;

- $array(\tau, n) \in LogTypeSpecs$, если $\tau \in LogTypeSpecs$, n — положительное целое число.

6. функции: $\tau \mapsto \tau'$;
7. декартовы произведения: $\tau \times \tau'$.

Алфавит. Алфавит языка утверждений состоит из следующих классов символов:

- переменные;
- константы (в частности, символы операций языка C-kernel);
- кванторы $\exists$ и $\forall$ и логические связки $\neg, \Rightarrow, \Leftrightarrow, \wedge, \vee$;
- скобки (,), [,], $<$, $>$, {, };
- символы пунктуации: точка, двоеточие, запятая.

Выражения. Переменные и константы могут быть любых типов, кроме пустого. Все переменные базовых типов помимо значений соответствующих типов могут иметь неопределённые значения, обозначаемые ω.

Выражения языка утверждений определяются по индукции следующим образом:

- переменная v типа τ является выражением типа τ;
- константа c типа τ является выражением типа τ;
- если $s_1, \ldots, s_n$ — выражения типов $\tau_1, \ldots, \tau_n$ соответственно, а s — выражение типа $\tau_1 \times \tau_2 \times \ldots \times \tau_n \mapsto \tau$, то $s(s_1, \ldots, s_n)$ является выражением типа τ.

Логические выражения строятся из выражений типа $\_Bool$ с помощью обычных логических связок и кванторов и для краткости не рассматриваются. Далее выражения типа $\_Bool$ называем просто утверждениями.

Важным свойством этого языка является то, что функциональные символы могут быть не только константами, но и переменными. То есть это не язык первого порядка.

4. Операционная семантика 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 [9], вычислительная машина должна быть абстрактной и не связанной с какой-либо конкретной архитектурой. Хотя память машины в нашем случае моделируется с высоким уровнем абстракции, для вычисления выражений выбрана конкретная стратегия. Это позволяет существенно упростить состояния машины по сравнению

с [11]. Тем не менее, описываемая операционная семантика придерживается стандарта в отношении многих других неспецифицированных аспектов языка C.

Абстрактная машина языка C-light. Определим состояния абстрактной машины, описывающей операционную семантику языка C-light. Пусть ID — это множество всех допустимых идентификаторов языка C. Далее для обозначения того, что элемент c не принадлежит области определения отображения ϕ, используем запись $\phi(c) = \perp$.

Определение. Состояние абстрактной машины языка C-light — это отображение, означающее следующие переменные:

1. MD — переменная типа $Locations \mapsto CTypes$.
2. STD — переменная типа $Names \mapsto LogTypeSpecs$.
3. Val — переменная типа $CTypes \times LogTypeSpecs$.
4. Переменные типа $Names$.

Метапеременная MD (Memory Dump) определяет значения, хранимые в памяти. Если мы хотим выразить тот факт, что значение некоторой целочисленной переменной x равно 5, то мы должны будем записать это так: $MD(\&x) = 5$. Несмотря на то, что мы ввели множество адресов в памяти $Locations$, побайтовая и, тем более, побитовая раскладка памяти не моделируется. Это связано с особенностями стандарта языка C и возможными существенными различиями конкретных реализаций его компиляторов. Стандарт не определяет ни размеры значений, ни внутреннее представление, ни размер байта. Поэтому будем считать, что базовые типы занимают в памяти "единичные" области, т. е. доступ к объекту может быть осуществлён только по его адресу как к единому целому, а доступ к отдельным байтам и битам запрещён.

Метапеременная STD сопоставляет идентификатору тип, связанный с ним в $typedef$-декларации или в декларации структуры или перечисления.

Метапеременная Val определяет возвращаемое функцией или выражением значение и его тип. Данная метапеременная используется для формализации оператора $return$.

Для обозначения множества всех состояний используем слово $States$. Для обозначения состояний используются греческие буквы σ, τ и другие с индексами и без. Иногда для удобства состояние будет записываться как индекс у переменной, и запись $MD_\sigma(c)$ есть краткая форма для $(\sigma(MD))(c)$.

Абстрактные функции. При работе абстрактной машины языка C-light используются специальные абстрактные функции.

Язык C-light наследует операции языка C. Для того чтобы задать вычисления этих операций в выражениях символически (не привязываясь к конкретной реализации языка C), используются функции $UnOpSem$ и $BinOpSem$ [11]. Функция $UnOpSem(\bullet, v, \tau)$ возвращает результат применения унарной операции $\bullet$ к значению v типа τ. Функция $BinOpSem(\circ, v_1, \tau_1, v_2, \tau_2)$ возвращает результат применения бинарной операции $\circ$ к значениям v_1 и v_2 типов τ_1 и τ_2 соответственно. Результаты этих функций имеют вид $Val(v', \tau')$, где v' — значение, τ' — его тип.

Функция $mb(e, id)$ вычисляет адрес элемента $e[id]$ массива e или поля $e.id$ структуры e — $mb : ID \times (N \cup ID) \mapsto Locations$. Она также не привязана к конкретной модели памяти.

Функция $cast(e, \tau, \tau')$ преобразует значение типа τ к значению типа τ'.

Функция $labels(S)$ возвращает множество меток, встречающихся на верхнем уровне в последовательности операторов S, за исключением $case$ и $default$-меток.

Функция $defaultValue(\tau, storage)$ возвращает значение по умолчанию для типа τ в соответствии с классом памяти $storage$:

- $defaultValue(\tau, static)$ = значение по умолчанию для типа τ;
- $defaultValue(\tau, auto) = \omega$.

Функции $id(e)$ и $tp(e)$ возвращают соответственно идентификатор и тип объекта, который объявляется в декларации e.

Декларация $e = e'$; находится в нормальной форме, если e' — инициализатор с полностью расставленными скобками, не содержащий выделенных инициализаторов, и объект, объявленный в e, имеет полный тип. Декларация $d = d'$; является нормальной формой декларации $e = e'$; если $d = d'$ находится в нормальной форме и эти декларации инициализируют один и тот же объект одним и тем же значением. Согласно [9] любую декларацию можно привести к нормальной форме. Булевская функция $isNform(d = d'; , e = e';)$ принимает значение $true$ тогда и только тогда, когда декларация $d = d'$; является нормальной формой декларации $e = e'$;.

Функция $storage(e)$ возвращает спецификатор класса памяти в декларации e:

- $storage(e) = auto$, если e содержит спецификатор $auto$ или $register$;
- $storage(e) = static$, если e содержит спецификатор $static$;
- $storage(e) = static$, если e не содержит спецификаторов класса памяти и e находится на внешнем уровне программы;
- $storage(e) = auto$, если e не содержит спецификаторов класса памяти и e находится не на внешнем уровне программы.

Пусть $A = (A_1, A_2)$. Функции $fst(A)$ и $snd(A)$ являются проекциями A на первую и вторую компоненты соответственно: $fst(A) = A_1$, $snd(A) = A_2$.

Семейство функций $type$ вычисляет тип конструкций языка C-light. Функция $type(\bullet, \tau_1, \tau_2)$ вычисляет тип значения, возвращаемого операцией $\bullet$ в случае, когда она имеет аргументы типов τ_1 и τ_2. Необязательный аргумент τ_2 задаётся для бинарных операций и условной операции. Семантика функции определяется в соответствии со стандартом ISO [9].

Пусть x — переменная, c — константа. Функция $type(e)$ вычисляет тип выражения e:

- $type(x) = \tau$, если существует декларация τx;;
- $type(c)$ = тип константы c;

- $type(\bullet e) = type(\bullet, (type(e))$;
- $type(e_1 \circ e_2) = type(\circ, type(e_1), type(e_2))$;
- $type(e_1?e_2 : e_3) = type(? :, type(e_2), type(e_3))$;
- $type((e)) = type(e)$;
- $type(e(e_1, \ldots, e_n)) = \tau$, если $type(e) = \tau_1 \times \tau_2 \times \ldots \times \tau_n \mapsto \tau$.

Функция $logtype(\tau, MD, STD)$ преобразует абстрактное имя типа τ в соответствующее ему логическое имя типа согласно значениям метапеременных MD и STD, модифицируя, если необходимо, последнюю: $logtype(\tau, MD, STD) = (\tau'$, $STD')$, где τ' — логическое имя типа, соответствующего абстрактному имени типа τ, STD' — результат модификации метапеременной STD.

Функция $addr(e, MD)$ вычисляет адрес выражения e в соответствии со значением метапеременной MD:

- $addr(e, MD) = \&e$, если e — переменная, к значению которой есть обращение через указатель;
- $addr(e, MD) = \omega$, если e — константа или переменная, к значению которой нет обращения через указатели;
- $addr(e[e'], MD) = \omega$, если e — массив, к значениям элементов которого нет обращения через указатели;
- $addr(e[e'], MD) = mb(e, val(e', MD, STD))$, в противном случае;
- $addr(e.m, MD) = \omega$, если e — структура, к значениям полей которой нет обращения через указатели;
- $addr(e.m, MD) = mb(val(e, MD, STD), val(m, MD, STD))$, в противном случае;
- $addr(\&e, MD) = \omega$;
- $addr(*e, MD) = val(e, MD, STD)$.

Переменные, к значениям которых нет обращения через указатели, будем называть паскалевскими.

Функция $val(e, MD, STD)$ вычисляет значение выражения e в соответствии со значениями метапеременных MD и STD:

- $val(e, MD, STD) = MD(\&e)$, если e — обычная переменная;
- $val(e, MD, STD) = e$, если e — константа или переменная, к значению которой нет обращения через указатели;
- $val(e[e'], MD, STD) = e[val(e', MD, STD)]$, если e — массив, к значениям элементов которого нет обращения через указатели;

- $val(e[e'], MD, STD) = MD(mb(e, val(e', MD, STD)))$, в противном случае;
- $val(e.m, MD, STD) = e.m$, если e — структура, к значениям полей которой нет обращения через указатели;
- $val(e.m, MD, STD) = MD(mb(val(e, MD, STD), val(m, MD, STD)))$, в противном случае;
- $val(\&e, MD, STD) = addr(e, MD)$;
- $val(*e, MD, STD) = MD(val(e, MD, STD))$;
- $val((\tau)e, MD, STD) = cast(val(e, MD, STD), type(e),$ $fst(logtype(\tau, MD, STD)))$;
- $val(\bullet e, MD, STD) = UnOpSem(\bullet, val(e, MD, STD), type(e))$, где $\bullet$ — унарная операция;
- $val(e \circ e', MD, STD) = BinOpSem(\circ, val(e, MD, STD), type(e),$ $val(e', MD, STD), type(e'))$, где $\circ$ — бинарная операция.

Определение. Конфигурация абстрактной машины языка C-light — это пара $\langle P, \sigma \rangle$, где P — программа, σ — состояние.

Аксиомы операционной семантики имеют вид $\langle A, \sigma \rangle \to \langle B, \sigma' \rangle$. Эта запись означает, что один шаг исполнения программного фрагмента A, начинающийся в состоянии σ, приводит в состояние σ' и B — фрагмент исходной программы, который остаётся для исполнения. Правила семантики имеют вид $\frac{P_1,\ldots,P_n}{\langle A,\sigma\rangle \to \langle B,\sigma'\rangle}$. Это означает, что при выполнении условий $P_1, \ldots, P_n$ можно перейти от конфигурации $\langle A, \sigma \rangle$ к конфигурации $\langle B, \sigma' \rangle$. Таким образом, исполнение программ в операционной семантике приводит к цепочками конфигураций, которые в общем случае могут быть и бесконечными:

$$\langle S_1, \sigma_1 \rangle \to \langle S_2, \sigma_2 \rangle \to \ldots \langle S_i, \sigma_i \rangle \to \ldots$$

Конфигурация $\langle S_1, \sigma_1 \rangle$ называется начальной. Если же цепочка конечна и $\langle S_n, \sigma_n \rangle$ — последняя конфигурация в этой цепочке, то говорим, что исполнение фрагмента S_1, начинающееся в состоянии σ_1 приводит в заключительную конфигурацию $\langle S_n, \sigma_n \rangle$. Если обозначить транзитивное рефлексивное замыкание отношения $\to$ как $\to^*$, то в общем случае такое исполнение можно обозначить как $\langle S_1, \sigma_1 \rangle \to^* \langle S_n, \sigma_n \rangle$.

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

Присваивание. Пусть τ'' — тип, отличный от массива, а e — не паскалевская переменная. Правило для простого присваивания имеет вид

$$\frac{\langle e_0, \sigma \rangle \to^* \langle Val(v', \tau'), \sigma' \rangle, \langle e, \sigma' \rangle \to^* \langle Val((v'', c), \tau''), \sigma'' \rangle, v = cast(val(v'), \tau', \tau'')}{\langle e = e_0, \sigma \rangle \to \langle Val(v, \tau''), \sigma''(MD \leftarrow upd(MD, c, v)) \rangle}$$

В случае, когда e — паскалевская переменная, правило принимает вид

$$\frac{\langle e_0, \sigma \rangle \to^* \langle Val(v', \tau'), \sigma' \rangle, \langle e, \sigma' \rangle \to^* \langle Val(e, \tau''), \sigma'' \rangle, v = cast(val(v'), \tau', \tau'')}{\langle e = e_0, \sigma \rangle \to \langle Val(v, \tau''), \sigma''(e \leftarrow v) \rangle}$$

Декларации переменных. Семантику декларации переменной определяют три правила — одно для декларации без инициализатора и два для декларации с инициализатором.

Пусть $e;$ — декларация переменной, не включающая спецификатора *enum* и имеющая единственный декларатор без инициализатора. Правило для декларации переменной без инициализатора имеет вид:

$$\frac{(\tau, STD') \in logtype(e, MD_\sigma, STD_\sigma), MD_\sigma(nc) = \bot, \quad (MD', V) \in init(\tau, storage(e), upd(MD_\sigma, nc, \omega))}{\langle e;\ , \sigma\rangle \to \langle \epsilon, \sigma(MD \leftarrow upd(MD', nc, fst(V)))(STD \leftarrow STD')\rangle}$$

Пусть $e = e';$ — декларация переменной, не включающая спецификатора enum и имеющая единственный декларатор с инициализатором. Правила для декларации переменной с инициализатором имеют вид:

$$\frac{isNform(d = d';, e = e';), \langle computeInit(d'), \sigma\rangle \to \langle Val(v), \sigma'\rangle, v \neq \omega, MD_\sigma(nc) = \bot, \quad (\tau, STD') \in logtype(d, MD_\sigma, STD_\sigma), (MD', V) \in init(\tau, v, upd(MD_\sigma, nc, \omega))}{\langle e = e';\ , \sigma\rangle \to \langle \epsilon, \sigma(MD \leftarrow upd(MD', nc, fst(V)))(STD \leftarrow STD')\rangle}$$

$$\frac{isNform(d = d';, e = e';), \langle computeInit(d'), \sigma\rangle \to \langle Val(\omega), \sigma'\rangle}{\langle e = e';\ , \sigma\rangle \to \langle Val(\omega), \sigma'\rangle}$$

Вспомогательная конструкция $computeInit(e)$ вычисляет значения всех выражений, входящих в инициализатор e, слева направо:

$$\frac{\langle computeInit(e_1), \sigma_0\rangle \to \langle Val(v_1), \sigma_1\rangle, v_1 \neq \omega \quad \dots \quad \langle computeInit(e_k), \sigma_{k-1}\rangle \to \langle Val(v_k), \sigma_k\rangle, v_k \neq \omega}{\langle computeInit(\{e_1, \dots, e_k\}), \sigma_0\rangle \to \langle Val(\{v_1, \dots, v_k\}), \sigma_k\rangle}$$

$$\frac{\langle computeInit(e_1), \sigma_0\rangle \to \langle Val(v_1), \sigma_1\rangle, v_1 \neq \omega \quad \dots \quad \langle computeInit(e_m), \sigma_{m-1}\rangle \to \langle Val(v_m), \sigma_m\rangle, v_m \neq \omega, m < k \quad \langle computeInit(e_{m+1}), \sigma_m\rangle \to \langle Val(v_k), \sigma_k\rangle, v_{m+1} = \omega}{\langle computeInit(\{e_1, \dots, e_k\}), \sigma_0\rangle \to \langle Val(\omega), \sigma_k\rangle}$$

Пусть инициализатор e не заключён в скобки. Тогда

$$\langle computeInit(e), \sigma\rangle \to \langle e, \sigma\rangle$$

Помеченный оператор. Помимо обычных меток операторы могут помечаться *case* и $default$-метками.

Оператор, помеченный меткой L, выполняется либо при нормальном последовательном выполнении программы, либо когда он ловит исключение $Exc(gotoStart(L))$, посланное оператором *goto* L:

$$\langle L : T, \sigma\rangle \to \langle T, \sigma\rangle$$

$$\langle Exc(gotoStart(L))\ L : T, \sigma\rangle \to \langle T, \sigma\rangle$$

Оператор, помеченный меткой *case*, выполняется либо при нормальном последовательном выполнении программы, либо когда он ловит исключение $Exc(switchStart(c, \tau))$, посланное оператором *switch*:

$$\langle case(e) : T, \sigma\rangle \to \langle T, \sigma\rangle$$

$$\frac{cast(val(e, MD_\sigma, STD_\sigma), type(e, MD_\sigma, STD_\sigma), \tau) = c}{\langle Exc(switchStart(c, \tau))\ case\ e : T, \sigma_0\rangle \to \langle T, \sigma\rangle}$$

Оператор, помеченный меткой $default$, выполняется либо при нормальном последовательном выполнении программы, либо когда он ловит исключение $Exc(defaultStart)$, посланное вспомогательной конструкцией *switchStop*:

$$\langle default : T, \sigma\rangle \to \langle T, \sigma\rangle$$

$$\langle Exc(defaultStart)\ default : T, \sigma\rangle \to \langle T, \sigma\rangle$$

Вспомогательная конструкция $switchStop(T)$ ловит исключение $Exc(switch\ Start(c))$, посланное оператором *switch* с телом T, в том случае, когда ни одна метка *case*, входящая в тело T, не совпала со значением управляющего выражения, и начинает выполнять тело T, сначала посылая исключение $Exc(defaultStart)$. В противном случае эта инструкция игнорируется:

$$\langle Exc(switchStart(c))\ switchStop(T)\ T', \sigma\rangle \to \langle Exc(defaultStart)\ T\ defaultStop\ T', \sigma\rangle$$

$$\langle switchStop(T), \sigma\rangle \to \langle \epsilon, \sigma\rangle$$

Вспомогательная конструкция $defaultStop$ ловит исключение $Exc(default\text{-}Start)$, посланное оператором $switchStop(T)$ в том случае, когда T не содержит метки $default$. В противном случае эта конструкция игнорируется:

$$\langle Exc(defaultStart)\ defaultStop\ T, \sigma\rangle \to \langle T, \sigma\rangle$$

$$\langle defaultStop, \sigma\rangle \to \langle \epsilon, \sigma\rangle$$

Блок. При определении семантики блока необходимо учитывать, что правила для оператора *goto* передают управление только вперёд. Встретив *goto* L, мы генерируем исключение $Exc(gotoStart(L))$ и пропускаем последующие операторы, пока не встретим оператор, помеченный меткой L, который ловит это исключение. Но как восстановить уже пройденную часть программы, если управление передаётся назад? Идея состоит в том, что при передаче управления мы всегда идём вперёд, но в конце каждого блока проверяем, принадлежит ли метка L блоку. Пусть метка L принадлежит блоку. Это означает, что в теле блока был встречен оператор *goto* L и метка находится выше по тексту. Тогда мы переходим в начало блока и спускаемся до метки L. Пусть метка L не принадлежит блоку. Тогда управление передаётся в охватывающий блок, в котором проводится точно такая же проверка. Для обработки блоков меток используется вспомогательная конструкция $gotoStop(T)$, где T — последовательность операторов блока. Аксиома для блока имеет вид:

$$\langle \{T\}, \sigma\rangle \to \langle T\ gotoStop(T), \sigma\rangle$$

Вспомогательная конструкция $gotoStop(T)$ ловит исключение $Exc(gotoStart(L))$, посланное оператором *goto* L в случае, когда метка L встречается в последовательности операторов T. В противном случае эта конструкция игнорируется:

$$\frac{L \in labels(T)}{\langle Exc(gotoStart(L))\, gotoStop(T)\, T', \sigma\rangle \to \langle Exc(gotoStart(L))\, T\, gotoStop(T)\, T', \sigma\rangle}$$

$$\langle gotoStop(T), \sigma\rangle \to \langle \epsilon, \sigma\rangle$$

Операторы выбора. Выполнение условного оператора определяется следующими правилами:

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(\omega), \sigma'\rangle}{\langle if\ (e)\, S_1\, else\, S_2, \sigma\rangle \to \langle Val(\omega), \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(v, \tau), \sigma'\rangle\ cast(val(v), \tau, int) \neq 0}{\langle if\ (e)\, S_1\, else\, S_2, \sigma\rangle \to \langle S_1, \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(v, \tau), \sigma'\rangle\ cast(val(v), \tau, int) = 0}{\langle if\ (e)\, S_1\, else\, S_2, \sigma\rangle \to \langle S_2, \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(\omega), \sigma'\rangle}{\langle if\ (e)\, S, \sigma\rangle \to \langle Val(\omega), \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(v, \tau), \sigma'\rangle\ cast(val(v), \tau, int) \neq 0}{\langle if\ (e)\, S, \sigma\rangle \to \langle S, \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(v, \tau), \sigma'\rangle\ cast(val(v), \tau, int) = 0}{\langle if\ (e)\, S, \sigma\rangle \to \langle \epsilon, \sigma'\rangle}$$

Оператор $switch\,(e)\,S$ вычисляет значение выражения e и посылает исключение $Exc(switchStart(c))$, которое может быть поймано операторами из S, помеченными метками *case* и $default$ в соответствии с правилами для помеченных операторов. Использование исключения позволяет не искать нужную ветвь в операторе *switch*, а непосредственно переходить к его телу:

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(\omega), \sigma'\rangle}{\langle switch\,(e)\,S, \sigma\rangle \to \langle Val(\omega), \sigma'\rangle}$$

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(\omega), \sigma'\rangle}{\begin{array}{c}\langle switch\,(e)\,S, \sigma\rangle \to \\ \langle Exc(switchStart((val(v), \tau)))\, S\, switchStop(S)\, gotoStop(S)\, breakStop, \sigma'\rangle\end{array}}$$

Вспомогательная конструкция *breakStop* ловит исключение $Exc(breakStart(c))$, посланное оператором *break*:

$$\langle Exc(breakStart)\, breakStop\, T, \sigma\rangle \to \langle T, \sigma\rangle$$

Циклы. В правилах для циклов используется вспомогательная конструкция $continueStop(e, S)$, которая ловит исключение $Exc(continueStart)$, посылаемое оператором *continue*, и, кроме этого, проверяет условие e цикла с телом S при выходе из цикла:

$$\frac{\langle e, \sigma\rangle \to^* \langle Val(\omega), \sigma'\rangle}{\langle continueStop(e, T)\, T', \sigma\rangle \to \langle Val(\omega), \sigma'\rangle}$$

$$\frac{\langle e,\sigma\rangle \to^* \langle Val(\omega),\sigma'\rangle}{\langle Exc(continueStart)\, continueStop(e,T)\, T',\sigma\rangle \to \langle Val(\omega),\sigma'\rangle}$$

$$\frac{\langle e,\sigma\rangle \to^* \langle Val(v,\tau),\sigma'\rangle\, cast(val(v),int) = 0}{\langle continueStop(e,T)\, T',\sigma\rangle \to \langle T',\sigma'\rangle}$$

$$\frac{\langle e,\sigma\rangle \to^* \langle Val(v,\tau),\sigma'\rangle\, cast(val(v),int) = 0}{\langle Exc(continueStart)\, continueStop(e,T)\, T',\sigma\rangle \to \langle T',\sigma'\rangle}$$

$$\frac{\langle e,\sigma\rangle \to^* \langle Val(v,\tau),\sigma'\rangle\, cast(val(v),int) \neq 0}{\langle continueStop(e,T)\, T',\sigma\rangle \to \langle T\, gotoStop(T)\, continueStop(e,T)\, T',\sigma'\rangle}$$

$$\frac{\langle e,\sigma\rangle \to^* \langle Val(v,\tau),\sigma'\rangle\, cast(val(v),int) \neq 0}{\langle Exc(continueStart)\, continueStop(e,T)\, T',\sigma\rangle \to \langle T\, gotoStop(T)\, continueStop(e,T)\, T',\sigma'\rangle}$$

При нормальном последовательном выполнении программы выполнение операторов цикла сводится к выполнению конструкции *continueStop*:

$$\langle while\,(e)\, S,\sigma\rangle \to \langle continueStop(e,S)\, breakStop,\sigma\rangle$$

$$\langle do\, S\, while\,(e),\sigma\rangle \to \langle S\, gotoStop(S)\, continueStop(e,S)\, breakStop,\sigma\rangle$$

$$\langle for\,(e_1;e_2;e_3)\, S,\sigma\rangle \to$$
$$\langle e_1; if\,(!e_2)\, break; S\, gotoStop(S)\, continueStop((e_3,e_2),S)\, breakStop,\sigma\rangle$$

Операторы перехода. Операторы перехода порождают исключения вида $Exc(\ldots)$. Эти исключения перехватываются другими конструкциями языка по аналогии с механизмом обработки исключений:

$$\langle goto\, L;,\sigma\rangle \to \langle Exc(gotoStart(L)),\sigma\rangle$$

$$\langle break;,\sigma\rangle \to \langle Exc(breakStart),\sigma\rangle$$

$$\langle continue;,\sigma\rangle \to \langle Exc(continueStart),\sigma\rangle$$

$$\langle return;,\sigma\rangle \to \langle Exc(returnStart),\sigma\rangle$$

$$\frac{\langle e,\sigma\rangle \to^* \langle Val(v,\tau),\sigma'\rangle}{\langle return\, e;,\sigma\rangle \to \langle Exc(returnStart(v,\tau)),\sigma'\rangle}$$

$$\frac{\langle e,\sigma\rangle \to^* \langle Val(\omega),\sigma'\rangle}{\langle return\, e;,\sigma\rangle \to \langle Val(\omega),\sigma'\rangle}$$

Аксиомы просачивания исключительных значений. Эти аксиомы применяются всякий раз, когда не применима ни одна другая аксиома:

$$\langle Exc(e)\, E\, T,\sigma\rangle \to \langle Exc(e)\, T,\sigma\rangle$$

$$\langle Val(\omega)\, E\, T,\sigma\rangle \to \langle Val(\omega)\, E\, T,\sigma\rangle$$

$$\langle Val(v,\tau)\, E\, T,\sigma\rangle \to \langle Val(v,\tau)\, T,\sigma\rangle$$

Заметим, что аксиомы просачивания не накладывают метаограничение, упорядочивающее применение аксиом и правил вывода операционной семантики языка C-light. Они являются только удобным сокращением для множества аксиом, элементы которых определяются конкретизацией T, а именно, заданием первого оператора в последовательности T. Поскольку число операторов языка C-light конечно, это множество конечно.

Программа. Правило для программы $Prgm(T)$, состоящей из последовательности деклараций T с одной выделенной функцией $main$, имеет вид:

$$\frac{\langle E, \sigma\rangle \to^* \langle \epsilon, \sigma'\rangle \, MD_\sigma = (params, S)}{\langle Prgm(T), \sigma\rangle \to \langle \{S\}\, returnStop\ Cast(\tau_{main}), \sigma'\rangle}$$

где τ_{main} — тип возвращаемого функцией $main$ значения. В нашем подходе аргументы функции $main$ считаются глобальными переменными, значения которых задаются входным состоянием σ.

5. Смешанная аксиоматическая семантика языка C-kernel

При определении смешанной аксиоматической семантики MHSC (Mixed Hoare Semantics for C-kernel) языка C-kernel возникают три проблемы:

1. Необходимо помнить информацию о текущей функции, иначе не формализовать операторы передачи управления *goto* и $return$.

2. Обработка вызова функции в логике Хоара задаётся на базе индуктивной гипотезы о том, что функция удовлетворяет своим спецификациям. Особую важность эта гипотеза приобретает для рекурсивных функций.

3. Обработка оператора *goto* L задаётся на базе индуктивной гипотезы о том, что в контрольной точке программы, предшествующей оператору, помеченному меткой L, инвариант, приписанный этой метке, истинен.

Информация о текущей функции задаётся окружением E. Окружение E — это пятёрка (f, τ, B, bid, lab), где f — имя текущей функции, τ — тип значения, возвращаемого этой функцией, B — её тело, bid — уникальный идентификатор текущего обрабатываемого блока, lab — метка, на которую осуществляется переход по встреченному в процессе вывода оператору *goto*, либо идентификатор блока тела функции, иначе $lab = \omega$. Мы полагаем, что функция $main$ является текущей функцией не только для программных конструкций, входящих в её тело, но и для всей программы, поскольку в силу специфики этой функции выполнение программы начинается с её вызова.

Информация о пред- и постусловиях функции и об инвариантах помеченных операторов задаётся спецификацией $SP = (SP_{fun}, SP_{lab})$, включающей спецификацию функций SP_{fun} и спецификацию меток SP_{lab}. Спецификация функций SP_{fun} — это пара (SP_{pre}, SP_{post}) отображений SP_{pre} и SP_{post}, называемых спецификациями

пред- и постусловий функций соответственно. Спецификация предусловий функции SP_{pre} — это отображение из имён функций f в функции P_f от метапеременных. Утверждение $P_f(MD, STD)$ называется предусловием функции f. Спецификация постусловий функции SP_{post} — это отображение из имён функций f в функции Q_f от метапеременных. Утверждение $Q_f(MD, STD)$ называется постусловием функции f. Спецификация меток SP_{lab} — это отображение из меток в утверждения. Утверждение $SP_{lab}(L)$ называется инвариантом метки L.

Формула Ψ языка аннотаций может содержать спецификацию предусловий функции SP_{pre} и спецификацию постусловий функции SP_{post}. Пусть $SP_{fun} \models \Psi$ обозначает истинность формулы в любом состоянии при фиксированной семантике отображений SP_{pre} и SP_{post} такой, что $SP_{pre} = fst(SP_{fun})$ и $SP_{post} = snd(SP_{fun})$.

Пусть символы P и Q обозначают аннотации, A, S и T — последовательности операторов, окружение $E = (f, \tau, B, cur, \omega)$, а идентификатор всей программы равен $Prgm$.

Теперь мы можем определить смешанную аксиоматическую семантику MHSC языка C-kernel как исчисление троек Хоара с окружениями. Выводимость тройки $\{P\}\ S\ \{Q\}$ в системе MHSC в окружении E относительно спецификации SP обозначим как $E, SP \vdash_{MHSC} \{P\}\ S\ \{Q\}$. Термин "смешанная" означает то, что правила вывода с участием переменных, к значениям которых нет доступа через указатели, могут иметь специальный (более простой) вид.

Для получения однозначности вывода условий корректности используется стратегия прямого прослеживания, при которой вывод проводится по программе от начала к концу (т. е. слева направо), и при этом преобразуется предусловие программы посредством последовательной элиминации самых левых операторов.

Выводимость программы $Prgm(T)$, состоящей из последовательности деклараций T, обозначим $E \vdash_{MHSC} \{P\}\ Prgm(T)\ \{Q\}$.

Всюду далее символы $'$ и $''$, стоящие рядом с именем переменной, означают введение новой переменной (т. е. не встречавшейся в аннотациях до применения правила, в котором она вводится).

Из-за ограниченности объёма в работу включены наиболее характерные правила аксиоматической семантики. Полный список можно найти в [7].

Операция присваивания. Пусть выражение e_0 не содержит вызовов функций, операторов *new* и операторов приведения. Для этих операторов правил, специфичных для смешанной семантики, не возникает — мы имеем их полные версии, описанные в [7]. Правило вывода для операции присваивания имеет вид:

$$\frac{\begin{array}{c} E, SP \vdash \{\exists MD'\ P(MD \leftarrow MD') \wedge \\ (MD = upd(MD', addr(e, MD'), cast(val(e_0, MD', STD), \\ type(e_0), type(e))))\}A; \{Q\} \end{array}}{E, SP \vdash \{P\}\ \mathbf{e = e_0};\ A; \{Q\}}, \qquad (1)$$

где e не является переменной паскалевского вида. Правило представляет собой модификацию правила классической системы Хоара: подстановки осуществляются на метапеременных, и производится приведение типа выражения e_0 к типу e.

В случае, когда e — простая переменная паскалевского вида, правило имеет вид:

$$\frac{\begin{array}{c}E, SP \vdash \{\exists e'\ P(e \leftarrow e') \wedge \\ (e = cast(val(e_0(e \leftarrow e'), MD, STD)), type(e_0(e \leftarrow e')), type(e))))\}A; \{Q\}\end{array}}{E, SP \vdash \{P\}\, \mathbf{e} = \mathbf{e_0};\ A; \{Q\}} \tag{2}$$

Если $e = v[i]$ — элемент массива паскалевского вида, то

$$\frac{\begin{array}{c}E, SP \vdash \{\exists v'\ P(v \leftarrow v') \wedge v = upd(v', val(i, MD, STD), \\ cast(val(e_0(v \leftarrow v'), MD, STD)), type(e_0(v \leftarrow v')), type(v[i]))))\}A; \{Q\}\end{array}}{E, SP \vdash \{P\}\, \mathbf{v[i]} = \mathbf{e_0};\ A; \{Q\}}. \tag{3}$$

Для структур паскалевского вида правило выглядит аналогичным образом. Правила вывода для операций работы с динамической памятью *new* и *delete* можно найти в [6, 7].

Декларации переменных. Правило вывода для декларации переменной, не являющейся переменной паскалевского вида, без инициализатора имеет вид:

$$\frac{\begin{array}{c}E, SP \vdash \{\exists MD'\ \exists STD'\ \exists nc\ \exists \tau\ \exists V\ \exists MD''\ \exists STD''\ P(MD \leftarrow MD') \wedge \\ (\tau, STD') \in logtype(tp(e), MD', STD') \wedge MD'(nc) = \bot \wedge \\ (MD'', V) \in init(\tau, storage(e), upd(MD', nc, \omega)) \wedge MD = upd(MD'', nc, fst(V)) \wedge \\ STD = STD''\}A; \{Q\}\end{array}}{E, SP \vdash \{P\}\, \mathbf{e};\ A; \{Q\}}. \tag{4}$$

Для простой переменной e паскалевского вида правило имеет вид:

$$\frac{E, SP \vdash \{\exists e'\ P(e \leftarrow e') \wedge e = defaultValue(\tau, storage)(e \leftarrow e')\}A; \{Q\}}{E, SP \vdash \{P\}\, \mathbf{storage}\ \tau\ \mathbf{e};\ A; \{Q\}}. \tag{5}$$

Аналогичным образом выглядят правила для декларации массивов и структур паскалевского вида.

Правило вывода для декларации переменной, не являющейся переменной паскалевского вида, с инициализатором имеет вид:

$$\frac{\begin{array}{c}E, SP \vdash \{\exists MD'\ \exists STD'\ \exists nc\ \exists \tau\ \exists V\ \exists MD''\ \exists STD''\ P(MD \leftarrow MD') \wedge \\ (\tau, STD') \in logtype(tp(e), MD', STD') \wedge MD'(nc) = \bot \wedge \\ (MD'', V) \in init(\tau, e_0, upd(MD', nc, \omega)) \wedge MD = upd(MD'', nc, fst(V)) \wedge \\ STD = STD''\}A; \{Q\}\end{array}}{E, SP \vdash \{P\}\, \mathbf{e} = \mathbf{e_0};\ A; \{Q\}}. \tag{6}$$

Для простой переменной e паскалевского вида правило имеет вид:

$$\frac{E, SP \vdash \{\exists e'\ P(e \leftarrow e')\ \wedge e = e_0(e \leftarrow e')\}A; \{Q\}}{E, SP \vdash \{P\}\, \mathbf{storage}\ \tau\ \mathbf{e} = \mathbf{e_0};\ A; \{Q\}}. \tag{7}$$

Помеченный оператор. Правила для помеченного оператора имеют вид:

$$\frac{\begin{array}{c}(f, \tau, B, cur, L), SP \vdash \{P\}A; \{Q\} \\ (f, \tau, B, cur, \omega), SP \vdash \{SP_{lab}(L_1)\}S;\ A; \{Q\}\end{array}}{(f, \tau, B, cur, L), SP \vdash \{P\}\{SP_{lab}(L_1)\}L_1 : S;\ A; \{Q\}}\ L \neq L_1, \tag{8}$$

$$\frac{\begin{array}{c} SP_{fun} \models P \Rightarrow SP_{lab}(L) \\ (f, \tau, B, cur, \omega), SP \vdash \{SP_{lab}(L)\}S;\ A; \{Q\} \end{array}}{(f, \tau, B, cur, E_5), SP \vdash \{P\}\{SP_{lab}(L_1)\}L_1 : S;\ A; \{Q\}}\ E_5 = \omega \text{ или } E_5 = L. \quad (9)$$

Правило (8) соответствует случаю, когда ранее был встречен оператор *goto* L, а в настоящий момент обрабатывается оператор, помеченный меткой L_1, отличной от L. Это значит, что метка L ещё не достигнута, и мы должны обработать оставшуюся часть программы A, но при этом также произвести обработку этой же части, начав с инварианта метки L_1 в качестве предусловия. Правило (9) соответствует случаям, когда ранее не был встречен ни один оператор *goto*, либо встречен оператор *goto* L и обрабатывается оператор, помеченный L. Выводится условие корректности $P \Rightarrow SP_{lab}(L)$, и продолжается обработка оставшейся части программы.

Блок. Правила для блока имеют вид:

$$\frac{(f, \tau, B, id, E_5), SP \vdash \{P\}S;\ blockEnd(cur);\ A; \{Q\}}{(f, \tau, B, cur, E_5), SP \vdash \{P\}\{S\}_{id}\ A; \{Q\}}, \quad (10)$$

$$\frac{(f, \tau, B, cur, E_5), SP \vdash \{P\}A; \{Q\}}{(f, \tau, B, id, E_5), SP \vdash \{P\}\ blockEnd(cur);\ A; \{Q\}} id \neq E_5, \quad (11)$$

$$\frac{SP_{fun} \models P \Rightarrow Q}{(f, \tau, B, id(f), id(f)), SP \vdash \{P\}\ blockEnd(cur);\ A; \{Q\}}. \quad (12)$$

Правило (10) раскрывает скобки блока, помечая его конец специальной конструкцией *blockEnd*, которая элиминируется правилами (11) и (12). Правило (11) соответствует случаю обычного выхода из блока и обеспечивает просачивание метки L (если был встречен оператор *goto* L, но не найден оператор, помеченный этой меткой L) и идентификатора тела функции f (если не был встречен оператор *return*). В случае правила (12) мы нашли соответствующую закрывающую скобку для ранее встреченного оператора *return*: обработка тела функции завершена, порождается условие корректности.

Пустой оператор. Правила вывода для пустого оператора имеют вид:

$$\frac{E, SP \vdash \{P\}A; \{Q\}}{E, SP \vdash \{P\};\ A; \{Q\}}, \quad (13)$$

$$\frac{SP_{fun} \models P \Rightarrow Q}{(f, \tau, B, cur, \omega), SP \vdash \{P\}\ ; \{Q\}}, \quad (14)$$

$$\frac{SP_{fun} \models P \Rightarrow SP_{lab}(L)}{(f, \tau, B, cur, L), SP \vdash \{P\}\ ; \{Q\}}. \quad (15)$$

Правило (13) применяется в том случае, когда оставшаяся часть программы A непустая. Правило (14) соответствует случаю, когда мы достигли конца обрабатываемой части программы. Порождается условие корректности $P \Rightarrow Q$. В случае правила (15) мы достигли конца обрабатываемой части программы, но не встретили оператор, помеченный меткой L. Порождается условие корректности $P \Rightarrow SP_{lab}(L)$.

Условный оператор. Правило для условного оператора имеет вид:

$$\frac{\begin{array}{c}E, SP \vdash \{P \wedge cast(val(e, MD, STD), type(e), int) \neq 0\} S_1;\ A; \{Q\} \\ E, SP \vdash \{P \wedge cast(val(e, MD, STD), type(e), int) = 0\} S_2;\ A; \{Q\}\end{array}}{E, SP \vdash \{P\} if\ (e)\ S_1\ else\ S_2;\ A; \{Q\}}. \tag{16}$$

Правило в точности повторяет правило классической системы Хоара с учётом особенности представления логического типа в языке C (а значит, и языке C-kernel) и приведения типов.

Цикл. Для определения свойств цикла используется принцип индукции, т. е. предполагается, что на каждой итерации выполняется некоторое инвариантное свойство INV:

$$\frac{\begin{array}{c}SP_{fun} \models P \Rightarrow INV \\ E, SP \vdash \{INV \wedge cast(val(e, MD, STD), type(e), int) \neq 0\} S; \{INV\} \\ E, SP \vdash \{INV \wedge cast(val(e, MD, STD), type(e), int) = 0\} A; \{Q\}\end{array}}{E, SP \vdash \{P\}\{INV\} while\ (e)\ S;\ A; \{Q\}}. \tag{17}$$

Таким образом, данное правило порождает одно условие корректности вида $P \Rightarrow INV$, осуществляет переход к обработке тела цикла S (в этом случае условие цикла e должно быть истинно, т. е. не 0 в терминах C) и переход к обработке оставшейся части программы A (в этом случае условие e ложно, т. е. 0).

Операторы перехода. Правила для оператора *goto* имеют вид:

$$\frac{(f, \tau, B, cur, L), SP \vdash \{P\}\ A; \{Q\}}{(f, \tau, B, cur, \omega), SP \vdash \{P\}\ goto\ L;\ A; \{Q\}}, \tag{18}$$

$$\frac{(f, \tau, B, cur, L), SP \vdash \{P\}\ A; \{Q\}}{(f, \tau, B, cur, L), SP \vdash \{P\}\ T;\ A; \{Q\}}, \tag{19}$$

где T не является помеченным оператором и не является $blockEnd(cur)$. Правило (18) добавляет в окружение метку L с целью поиска соответствующего помеченного оператора, а правило (19) осуществляет её просачивание. Тем самым мы продолжаем двигаться по программе, пропуская операторы (кроме выхода из блока), пока не встретим оператор, помеченный меткой L.

Семантику оператора $return$ определяют три правила вывода: для $return$ с аргументом и $return$ без аргументов.

$$\frac{(f, \tau, B, cur, id(f)), SP \vdash \{P\}\ A; \{Q\}}{(f, \tau, B, cur, \omega), SP \vdash \{P\}\ return;\ A; \{Q\}}, \tag{20}$$

$$\frac{(f, \tau, B, cur, id(f)), SP \vdash \{P\}\ A; \{Q\}}{(f, \tau, B, cur, id(f)), SP \vdash \{P\}\ T;\ A; \{Q\}}, \tag{21}$$

$$\frac{\begin{array}{c}(f, \tau, B, cur, id(f)), SP \vdash \{\exists Val' P(Val \leftarrow Val') \wedge \\ Val = (cast(val(e, MD, STD), type(e), \tau), \tau)\}\ A; \{Q\}\end{array}}{(f, \tau, B, cur, \omega), SP \vdash \{P\}\ return\ e;\ A; \{Q\}}, \tag{22}$$

где T не является помеченным оператором и не является $blockEnd(\ldots)$. Правила (20) и (21) аналогичны правилам (18) и (19) для оператора *goto*, только вместо метки перехода в окружение добавляется идентификатор блока тела текущей функции с целью поиска соответствующей концу этого блока конструкции $blockEnd$. Правило (22) соответствует случаю, когда функция возвращает значение. В этом случае результат сохраняется в метапеременной Val.

Программа. Правило для программы $Prgm(T)$, состоящей из последовательности деклараций T, имеет вид:

$$\frac{\begin{array}{c}(f, \tau_f, S_f, bid_f, \omega), ((SP_{pre}, SP_{post}), SP^f_{lab}) \vdash \{SP_{pre}(f)(x_1, \ldots, x_n)(MD, STD, Val)\} \\ S_f;\ blockEnd(bid_f)\{SP_{post}(f)(x_1, \ldots, x_n)(MD, STD, Val)\} \\ \text{по всем именам функций } f, \text{ определённых в } T, \text{ кроме } main, \\ (main, \tau_{main}, S_{main}, bid_{main}, \omega), SP \vdash \{P\}\ T\ S_{main}; blockEnd(bid_{main})\{Q\}\end{array}}{E, SP \vdash \{P\}\ Prgm(T)\{Q\}}, \tag{23}$$

где $x_1, \ldots, x_n$ — формальные параметры функции f.

В системе MHSC так же есть правила консеквенции и для последовательности операторов, которые определяются стандартным образом.

6. Перевод инвариантов из C-light в C-kernel

При переводе программы на языке C-light в программу на языке C-kernel [1, 2] возникает проблема вычисления инвариантов циклов и меток полученной программы. При элиминации операторов *break* и *continue* внутри циклов и оператора *switch*, а также самого оператора *switch* возникают новые метки, каждой из которых в соответствии с правилами смешанной аксиоматической семантики языка C-kernel для помеченного оператора должен быть сопоставлен некоторый инвариант, который отсутствует в исходной программе. Рассмотрим решение данной проблемы.

Оператор цикла $do - while$ в соответствии с правилами перевода заменяется на оператор *while*:

$$\{INV\}\ do\ \{A\}\ while(e) \rightarrow \{INV'\}\ while(1)\ \{A\ if(e)\ \{\}else\ break;\},$$

а элиминация операторов *break* и *continue* в теле *while*-цикла происходит следующим образом:

$$while(e)\ \{A\ break;\ B\} \rightarrow \{while(e)\ \{A\ goto\ L;\ B\}\ \{INV(L)\}\ L:\},$$

$$while(e)\ \{A\ continue;\ B\} \rightarrow while(e)\ \{A\ goto\ L;\ B\ \{INV(L)\}\ L:\},$$

где L — новая метка, INV — инвариант цикла, INV' — искомый инвариант полученного цикла, $INV(L)$ — искомый инвариант метки L, которая появляется сразу после тела цикла. По определению инвариант цикла истинен до входа в цикл, на каждой его итерации и при выходе. Следовательно, $INV(L) \equiv INV \equiv INV'$.

for-цикл заменяется на цикл *while*:

$$\{INV\}\ for(e_1; e_2; e_3)\ \{A\} \rightarrow \{INV\}\ e_1; \{INV'\}\ while(e_2)\ \{A\ e_3\}$$

По определению инвариант истинен перед началом каждой итерации цикла, а инициализирующее выражение e_1 вычисляется только один раз. Поэтому $INV' \equiv INV$.

Оператор *continue* в теле for-цикла преобразуется следующим образом:

$$for(e_1; e_2; e_3)\ \{A;\ continue;\ B\} \to e_1;\ while(e_2)\ \{A;\ goto\ L;\ B\ \{INV(L)\}\ L : e_3\}$$

В данном случае оператор e_3 оказывается оператором, помеченным меткой L, и мы не можем использовать инвариант цикла, так как это не начало цикла, не начало новой итерации и не выход из цикла.

Аналогичная проблема возникает для оператора *switch* — при его элиминации возникают новые метки, число которых равно числу ветвей *case*:

$$\begin{gathered} switch(d)\ \{case\ c_1\ A_1; \ldots;\ case\ c_n\ A_n;\ default\ A_{n+1};\} \to \\ \{if(d == c_1)\ goto\ L_1; \ldots; if\ (d == c_n)\ goto\ L_n;\ goto\ L_{default}; \\ \{INV(L_1)\}\ L_1 : A_1;\ \ldots; \{INV(L_n)\}\ L_n : A_n;\ \{INV(L_{default})\}\ L_{default}\ A_{n+1}\} \end{gathered}$$

Кроме того, новая метка появляется и при элиминации оператора *break* в теле оператора *switch*:

$$switch(d)\ \{A\ break;\ B\} \to \{switch(d)\ \{A\ goto\ L;\ B\}\ \{INV(L)\}\ L :\}$$

Пусть после перевода программы из C-light в C-kernel появилось n неизвестных инвариантов меток. Обозначим их $INV(L_1), \ldots, INV(L_n)$. Пусть выведены все условия корректности. Тогда в силу правил (9), (12), (14), (15), (17) среди них будут такие:

$$\begin{array}{ll} P_0 \Rightarrow INV(L_1); & INV(L_1) \wedge P_{k_1+1} \Rightarrow INV(L_2); \\ P_1 \Rightarrow INV(L_1); & \ldots \\ \ldots & INV(L_1) \wedge P_{k_2} \Rightarrow INV(L_2); \\ P_{k_1} \Rightarrow INV(L_1); & \ldots \\ & INV(L_{n-1}) \wedge P_{k_n} \Rightarrow INV(L_n), \end{array}$$

где P_i — формулы, получающиеся по правилам вывода модифицированной аксиоматической семантики и не содержащие неизвестных подформул.

Разрешая данную систему относительно $INV(L_i)$ в предположении, что составляющие её условия корректности должны быть истинными одновременно, получим:

$$INV(L_i) \equiv (P_0 \vee P_1 \vee \ldots \vee P_{k_1}) \wedge (P_{k_1+1} \vee \ldots \vee P_{k_2}) \wedge \ldots \wedge (P_{k_{i-1}+1} \vee \ldots \vee P_{k_i})$$

Подставляя найденные значения инвариантов меток в остальные формулы, получим условия корректности без неизвестных подформул.

Таким образом, при переводе программы из C-light в C-kernel возникающие инварианты меток вычисляются из имеющейся спецификации исходной программы.

7. Пример

Рассмотрим программу на языке C-light, которая в заданном массиве находит первый по порядку индексов отрицательный элемент и заменяет его на равный по

модулю положительный. Несмотря на свою простоту, эта программа хорошо показывает особенности смешанной аксиоматической семантики по сравнению с полной семантикой, предложенной в [1].

Программа на C-light:	Программа на C-kernel:
/* SP_{pre}(NegateFirst) */	/* SP_{pre}(NegateFirst) */
void NegateFirst(void)	void NegateFirst(void)
{ int i;	{ static int i;
/* INV_{for} */	i = 0;
for (i = 0; i < lgt; i++)	/* INV_{while} */ while (i < lgt)
{ if (M[i] < 0)	{ if (M[i] < 0)
{ M[i] = -M[i];	{ M[i] = -M[i];
break; }}}	goto L; $\}_{then}$ else $\{\}_{else}$
/* SP_{post}(NegateFirst) */	i = i + 1; $\}_{while}$
	/* INV(L) */ L: $\}_{NegFirst}$
	/* SP_{post}(NegateFirst) */

В силу изложенного в разделе 6 имеем, что $INV_{while} \equiv INV_{for}$. Специфицируем программу.

1. Предусловие $SP_{pre}(NegateFirst) \equiv M = M_0 \wedge lgt = length(M_0)$ фиксирует начальные данные. Функция $length$ возвращает число элементов массива.

2. Инвариант цикла $INV_{for} \equiv \forall j(0 \leq j < i \Rightarrow M[j] \geq 0)$ утверждает, что все просмотренные элементы массива неотрицательны.

3. Постусловие $SP_{post}(NegateFirst) \equiv M = NegFst(M_0)$, где $NegFst(A)$ — массив, получающийся из массива A заменой первого по порядку индексов отрицательного элемента на равный по модулю положительный: $NegFst((a_1, a_2, \ldots, a_n)) = (a_1, a_2, \ldots, a_{k-1}, -a_k, a_{k+1}, \ldots, a_n)$, где $k = min\{1, \ldots, n | a_k < 0\}$.

Заметим, что в данной программе все переменные, в том числе и массив M, являются паскалевскими, поскольку доступа к их значению через указатели нет. Поэтому в процессе вывода условий корректности используются специальные упрощенные версии правил вывода для операции присваивания. В соответствии с правилом вывода для цикла первое условие корректности получается на участке от начала программы до начала цикла:

$$\exists i_2(\exists i_1(M = M_0 \wedge lgt = length(M_0) \wedge i_2 = \omega) \wedge i = 0) \Rightarrow \forall j(0 \leq j < i \Rightarrow M[j] \geq 0)$$

Истинность очевидна, поскольку не существует такого j, что $0 \leq j < 0$.

Далее переходим к рассмотрению тела цикла, внутри которого имеется условный оператор. В соответствии с правилом вывода для данного оператора (16) нужно рассмотреть обе ветви, но т. к. *else*-ветвь пуста, а далее имеется одна операция присваивания, то получаем следующее условие корректности:

$$\exists i_1(\forall j(0 \leq j < i_1 \Rightarrow M[j] \geq 0) \wedge i_1 < lgt \wedge M[i_1] \geq 0 \wedge i = i_1 + 1) \Rightarrow \forall j(0 \leq j < i \Rightarrow M[j] \geq 0)$$

Согласно посылке $\forall j(0 \le j < i_1 \Rightarrow M[j] \ge 0)$. Следовательно, $\forall j(0 \le j < i - 1 \Rightarrow M[j] \ge 0)$, но $i - 1 < i$, поэтому заключение истинно.

В ветви *then* условного оператора содержится оператор перехода *goto*. В соответствии с правилами для операторов перехода (18), (19) и блока (11) мы достигаем конца обрабатываемой части программы. Поэтому согласно правилу для пустого оператора (15) немедленно получаем условие корректности:

$$\exists M_1(\forall j(0 \le j < i \Rightarrow M_1[j] \ge 0) \wedge i < lgt \wedge M_1[i] < 0 \wedge M = upd(M_1, i, -M_1[i])) \Rightarrow INV(L)$$

Наконец, переходим к рассмотрению оставшейся части программы, которая работает после выхода из цикла: она состоит из пустого помеченного оператора. В соответствии с правилом (9) получаем условие корректности:

$$\forall j(0 \le j < i \Rightarrow M[j] \ge 0) \wedge i \ge lgt \Rightarrow INV(L)$$

Оставшийся пустой оператор даёт нам последнее условие корректности:

$$INV(L) \Rightarrow M = NegFst(M_0)$$

Последние три условия корректности содержат неизвестный инвариант $INV(L)$. Используя метод, изложенный в разделе 6, находим его:

$$INV(L) \equiv \exists M_1(\forall j(0 \le j < i \Rightarrow M_1[j] \ge 0) \wedge i < lgt \wedge M_1[i] < 0 \wedge M = upd(M_1, i, -M_1[i])) \vee \forall j(0 \le j < i \Rightarrow M[j] \ge 0) \wedge i \ge lgt$$

Таким образом, осталось показать, что

$$\exists M_1(\forall j(0 \le j < lgt \Rightarrow M_1[j] \ge 0) \wedge i < lgt \wedge M_1[i] < 0 \wedge M = upd(M_1, i, -M_1[i])) \vee \forall j(0 \le j < i \Rightarrow M[j] \ge 0) \wedge i \ge lgt \Rightarrow M = NegFst(M_0).$$

Предположим, что истинен первый дизъюнкт посылки, следовательно, существует такое i, что все $M_1[j]$, где j меняется от 0 до $i - 1$ неотрицательны, а $M_1[i]$ отрицателен. Значит, i – минимальный такой индекс и заключение истинно.

Пусть теперь истинен второй дизъюнкт посылки. В этом случае все элементы исходного массива неотрицательны и заключение очевидно. Тем самым рассмотренная программа частично корректна.

8. Заключение

Данная работа продолжает цикл статей, связанных с проектом лаборатории теоретического программирования ИСИ СО РАН по верификации программ на языке C-light. Предложенная смешанная аксиоматическая семантика MHSC является основой генератора условий корректности автоматизированной системы верификации C-light программ СПЕКТР-2. С её помощью был успешно верифицирован ряд программ, представляющих трудности для верификации. Среди них — программа топологической сортировки [7]. В дальнейшем предполагается разработать формальное обоснование корректности смешанной аксиоматической семантики MHSC относительно операционной.

Список литературы

1. Непомнящий В. А., Ануреев И. С., Михайлов И. Н., Промский А. В. Ориентированный на верификацию язык C-light // Системная информатика: сборник научных трудов. Новосибирск: Издательство СО РАН. 2004. Вып. 9: Формальные методы и модели информатики. С. 51 – 134.

2. Непомнящий В. А., Ануреев И. С., Промский А. В. На пути к верификации С-программ. Язык C-light и его трансформационная семантика // Problems in Programming. Kiev, 2006. № 2 – 3. С. 359 – 368.

3. Непомнящий В. А., Ануреев И. С., Михайлов И. Н., Промский А. В. На пути к верификации С программ. Язык C-light и его формальная семантика // Программирование. 2002. № 6. С. 19 – 30.

4. Непомнящий В. А., Ануреев И. С., Михайлов И. Н., Промский А. В. На пути к верификации С программ. Аксиоматическая семантика языка C-kernel // Программирование. 2003. № 6. С. 5 – 15.

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. Марьясов И. В. На пути к автоматической верификации C-light программ. Смешанная аксиоматическая семантика языка C-kernel. Новосибирск, 2008. 32 с. (Препр. / РАН. Сиб. Отд-ние. ИСИ; № 150).

7. Марьясов И. В. Применение смешанной аксиоматической семантики языка C-kernel к верификации программы топологической сортировки. – Новосибирск, 2010. – 33 с. – (Препр. / РАН. Сиб. Отд-ние. ИСИ; № 155).

8. Непомнящий В. А., Рякин, О. М. Прикладные методы верификации программ. М.: Радио и связь, 1988.

9. Programming languages – C: ISO/IEC 9899:1999. 1999. 566 p.

10. Maryasov I. V. Towards automatic verification of C-light programs. Mixed axiomatic semantics of C-kernel language // Perspectives of Systems Informatics (PSI): A. Ershov 7th Int. Conf.: Int. workshop on Program Understanding. Novosibirsk, 2009. P. 44 – 52.

11. Norrish M. C formalised in HOL: Thes. doct. phylosophy (computer sci.). Cambridge, 1998. 150 p.

C-programs Verification on Basis of Mixed Axiomatic Semantics

Anureev I. S., Maryasov I. V., Nepomniaschy V. A.

Keywords: program verification, operational semantics, axiomatic semantics, C language, C-light language, C-kernel language, partial correctness

The mixed axiomatic semantics of a C-kernel language is described. This language is the kernel of a representative C language subset which is called C-light. Such semantics allows to simplify the verification conditions in many cases. This semantics is a base of verification conditions generator of C-kernel programs. An example which illustrates the use of inference rules of the semantics is considered.

Сведения об авторах:
Ануреев Игорь Сергеевич, ИСИ СО РАН, ст. науч. сотр.;
Марьясов Илья Владимирович, ИСИ СО РАН, мл. науч. сотр.;
Непомнящий Валерий Александрович, ИСИ СО РАН, зав. лаб., НГУ, доцент