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

И. В. Марьясов

НА ПУТИ К АВТОМАТИЧЕСКОЙ ВЕРИФИКАЦИИ ПРОГРАММ НА ЯЗЫКЕ C-LIGHT. СМЕШАННАЯ АКСИОМАТИЧЕСКАЯ СЕМАНТИКА ЯЗЫКА C-KERNEL

Препринт
150

Новосибирск 2008

В настоящей работе описывается смешанная аксиоматическая семантика языка C-kernel, являющегося ядром ориентированного на верификацию языка C-light. Данная семантика является основой разрабатываемого генератора условий корректности C-kernel-программ. Рассмотрено несколько примеров, иллюстрирующих применение правил вывода описанной семантики.

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

I. V. Maryasov

TOWARDS THE AUTOMATIC VERIFICATION OF C-LIGHT PROGRAMS. MIXED MODIFIED AXIOMATIC SEMANTICS OF C-KERNEL LANGUAGE

Preprint
150

Novosibirsk 2008

A mixed modified axiomatic semantics of C-kernel language is described. This language is a kernel of C-light language oriented to verification. This semantics is a basement of verification conditions generator of C-kernel programs. Several examples which illustrate the use of inference rules are considered.

1. ВВЕДЕНИЕ

В настоящее время в лаборатории теоретического программирования ИСИ СО РАН ведётся работа над несколькими проектами по верификации программ, написанных на различных языках программирования. Один из таких языков — C-light, являющийся представительным подмножеством языка C. С целью избежания громоздкости аксиоматической семантики в языке C-light было выделено ядро — язык C-kernel, для которого была разработана аксиоматическая семантика и в который транслируются исходные программы на языке C-light. По сравнению с языком C-light в нём более простые выражения и более ограниченный набор операторов. Однако предложенная семантика не ориентирована на автоматизацию процесса вывода условий корректности, поскольку она сконструирована для отдельных операторов независимо от контекста. Кроме того, условия корректности даже для небольших программ получались громоздкими, что затрудняло их последующее доказательство.

В данной работе предлагается модифицированная смешанная аксиоматическая семантика языка C-kernel. Термин «модифицированная» означает, что семантика обладает однозначностью вывода (т. е. на каждом шаге анализа в ней допустимо одно и только одно правило вывода), а «смешанная» — что в неё введены правила вывода специального вида, применяемые, когда в анализируемом фрагменте программы нет переменных, доступ к значению которых осуществляется через указатели. Это позволяет во многих случаях существенно упростить условия корректности.

Работа имеет следующую структуру. В разд. 2 описан язык утверждений, используемый для описания свойств программы. Разд. 3 посвящён правилам вывода модифицированной смешанной аксиоматической семантики MMHSC языка C-kernel. В разд. 4 рассмотрены примеры, иллюстрирующие применение правил описанной семантики.

2. ЯЗЫК УТВЕРЖДЕНИЙ

2.1. Типы и алфавит

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:

- $\tau \in \text{LogTypeSpecs}$, если τ — базовый тип;
- $\text{pointer}(\tau) \in \text{LogTypeSpecs}$, если $\tau \in \text{LogTypeSpecs}$;
- $\tau_1 \times \ldots \times \tau_n \to \tau \in \text{LogTypeSpecs}$, если $\tau_1, \ldots, \tau_n, \tau \in \text{LogTypeSpecs}$;
- $\text{struct}(s, (\tau_1, m_1), \ldots, (\tau_n, m_n)) \in \text{LogTypeSpecs}$, если $\tau_1, \ldots, \tau_n \in \text{LogTypeSpecs}$ и $s, m_1, \ldots, m_n \in \text{Names}$;
- $\text{enum}(s, m_1 \ldots, m_n) \in \text{LogTypeSpecs}$, если $s, m_1, \ldots, m_n \in \text{Names}$;
- $\text{array}(\tau, n) \in \text{LogTypeSpecs}$, если $\tau \in \text{LogTypeSpecs}$, n — положительное целое число.

6. функции: $\tau \to \tau'$;

7. декартовы произведения: $\tau \times \tau'$;

8. lv-типы $\text{LV}(\tau)$, служащие для хранения l-значений. Пусть v, c и d (возможно, с индексами) — значения типов τ, τ^* и $\text{LV}(\tau)$ соответственно. Тип $\text{LV}(\tau)$ задаётся как алгебраический тип данных с тремя операциями lv, val и loc, определяемыми следующими аксиомами:

$\forall v\ \forall c\ \text{val}(\text{lv}(v, c)) = v$,

$\forall v\ \forall c\ \text{loc}(\text{lv}(v, c)) = c$,

$\forall v\ \forall c\ \exists d\ d = \text{lv}(v, c)$,

$\forall d\ \exists v\ \exists c\ d = \text{lv}(v, c)$,

$\forall d\ \forall d'\ (d = d' \Rightarrow \text{val}(d) = \text{val}(d') \wedge \text{loc}(d) = \text{loc}(d'))$.

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.

2.2. Выражения

Переменные и константы могут быть любых типов, кроме пустого. Множество всех переменных обозначается идентификатором Var. Все переменные базовых типов, помимо значений соответствующих типов, могут иметь неопределённые значения. Неопределённое значение обозначается как 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 $\tau_1 \times \ldots \times \tau_n \to \tau$, 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.

Выражения языка утверждений имеют префиксный вид. В частности, это означает, что стандартные операции языка C-light в утверждениях записываются именно в такой форме. В примерах для удобства будем пользоваться обычной инфиксной формой.

2.3. Метапеременные

Для простого языка программирования, например, Паскаля, часто допустимо использовать в утверждениях имена программных переменных. Однако если в языке имеются указатели, которым можно присваивать адреса других объектов, то такой подход приведёт к проблемам при определении аксиоматической семантики. С другой стороны, если в программе встречаются переменные, к значению которых нет обращения через указатели, то, как и в классическом случае, допустимо, чтобы идентификаторы этих переменных в языке утверждения обозначали их значения в памяти. Другой особенностью языка C (а следовательно и языков C-light и C-kernel) являются typedef-декларации, вводящие синонимы допустимых типов. Всё это потребовало введение в язык утверждений *метапеременных*, описанных ниже.

Пусть ID — это множество всех допустимых идентификаторов языка C, Names — множество имён типов.

1. MeM — метапеременная типа ID → Locations, которая определяет адресацию объектов. Поскольку в языке C-kernel все идентификаторы уникальны, то отображение MeM (Memory Management) зависит только от имени переменной.

2. MD — метапеременная типа Locations → CTypes, которая определяет значения, хранимые в памяти. Зная адрес объекта, можно узнать значение, находящееся по этому адресу, с помощью отображения MD (Memory Dump). Если мы хотим выразить тот факт, что значение некоторой целочисленной переменной x равно 5, то мы должны будем записать это так: MD(MeM(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.

3. TP — метапеременная типа Locations → LogTypeSpecs, которая определяет типы значений в памяти. Поскольку в выражения могут входить объекты разных типов, то нужно их знать, чтобы осуществлять их приведение. Эту информацию и определяет отображение TP (TyPes). Например, для той же переменной x справедливо равенство TP(MeM(x))=int.

4. STD — метапеременная типа Names → LogTypeSpecs, которая определяет информацию о тегах и синонимах типов (typedef-декларации). Это отображение сопоставляет идентификатору тип, связанный с ним в typedef-декларации или в декларации структуры или перечисления.

5. Val — метапеременная типа CTypes × LogTypeSpecs, определяющая возвращаемое функцией или выражением значение и его тип. Данная метапеременная используется для формализации оператора return.

Всюду далее MeM..STD обозначает последовательность метапеременных (возможно штрихованных) MeM, MD, TP, STD.

Для произвольного отображения F стандартным образом определим *функцию обновления upd(F, a, b)*:

- upd(F, a, b)(a)=b;
- upd(F, a, b)(c)=F(c), если c≠a.

2.4. Абстрактные функции

Поскольку определение языка C-light было дано с помощью операционной семантики, то некоторые абстрактные функции, использованные при этом определении, будут использоваться и при задании аксиоматической семантики. Дадим подробное описание этих функций.

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.

Функция UnOpSem(•, v, τ) возвращает результат применения унарной операции • к значению v типа τ. Результат имеет вид Val(v', τ'), где v' — значение, τ' — его тип.

Функция BinOpSem(•, v_1, τ_1, v_2, τ_2) возвращает результат применения бинарной операции • к значениям v_1 и v_2 типов τ_1 и τ_2 соответственно. Результат имеет вид Val(v, τ), 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 × (N ∪ ID) → Locations.

Функция cast(e, τ, τ') приводит значение типа τ к значению типа τ'.

Функция findex(τ) возвращает первый индекс составного типа τ:

- findex(τ)=0, если τ — массив;
- findex(struct(s, (τ_1, l_1), …, (τ_k, l_k)))=l_1.

Функция next(τ, l) возвращает индекс составного типа τ, непосредственно следующий за индексом l:

- next(array(τ, k), l)=l+1, если l+1<k;
- next(array(τ, k), l)=ω, если l+1≥k;
- next(struct(s, (τ_1, l_1), …, (τ_k, l_k)), l_i)=l_{i+1}, если i<k;
- next(struct(s, (τ_1, l_1), …, (τ_k, l_k)), l_i)=l_{i+1}, если i≥k.

Функция id(e) возвращает идентификатор объекта, который объявляется в декларации e.

Функция tp(e) возвращает тип объекта, который объявляется в декларации e.

Функция storage(e) возвращает спецификатор класса памяти в декларации e.

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.

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

- defaultValue(τ, static)=значение по умолчанию для типа τ;
- defaultValue(τ, static)=ω.

Семейство *функций type* вычисляет тип конструкций языка C-light. Оно включает три функции.

Функция type(u), вычисляющая тип константы u, задаётся следующими аксиомами, которые разбиты на три группы.

Первая группа из девяти аксиом стандартным образом определяет типы числовых констант (восьмеричные и шестнадцатеричные константы для краткости не рассматриваются). Как обычно, для знаковых целых ключевое слово signed опускается:

- type(n)=один из {int, long int, long long int};
- type(nU)=один из {unsigned int, unsigned long int, unsigned long long int};
- type(nL)=один из {long int, long long int};
- type(nUL)=один из {unsigned long int, unsigned long long int};
- type(nLL)=long long int;
- type(nULL)=unsigned long long int;
- type([n_1].[n_2][E [±] n_3])=double;
- type([n_1].[n_2][E [±] n_3]F)=float;
- type([n_1].[n_2][E [±] n_3]L)=long double.

Фраза «один из» означает, что в качестве типа константы выбирается первый тип, достаточный для её хранения. Квадратными скобками обозначены необязательные элементы.

Вторая группа из четырёх аксиом задаёт типы символьных и строковых констант:

- type('c')=int;
- type(L'c_1c_2')=unsigned short;
- type("$c_1 \ldots c_n$")=array(char, n+1);
- type(L"$c_1 \ldots c_n$")=array(unsigned short, n/2+1).

Первая аксиома здесь отражает специфику языка C — символьные константы имеют целый тип (хотя символьные строки — это массивы типа char). В данной семантике не рассматриваются простые символьные константы, состоящие из более чем одного символа или содержащие нестандартные (то есть не соответствующие никакому однобайтному символу) escape-последовательности, поскольку их значения зависят от реализации.

Третья группа состоит из аксиомы для нулевого указателя: type(NULL)=void*.

Функция type(•, τ_1, τ_2) вычисляет тип значения, возвращаемого операцией • 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.

Пусть x — переменная, c — константа. Функция *type(e, MeM, TP)* вычисляет тип выражения e в соответствии со значениями метапеременных MeM и TP:

- type(x, MeM, TP)=TP(MeM(x));
- type(c, MeM, TP)=type(c);
- type(•e, MeM, TP)=type(•, (type(e, MeM, TP));
- type(e_1•e_2, MeM, TP)=type(•, type(e_1, MeM, TP), type(e_2, MeM, TP));
- type(e_1?e_2:e_3, MeM, TP)=type(? :, type(e_2, MeM, TP), type(e_3, MeM, TP));
- type((e), MeM, TP)=type(e, MeM, TP);
- type(e(e_1, …, e_n), MeM, TP)=τ, если type(e, MeM, TP)= $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$.

Функция logtype(τ, MeM..STD) преобразует абстрактное имя типа τ в соответствующее ему логическое имя типа согласно значениям метапеременных MeM, MD, TP и STD, модифицируя, если необходимо, последнюю:

logtype(τ, MeM..STD)=(τ', STD'),

где τ' — логическое имя типа, соответствующего абстрактному имени типа τ, STD' — результат модификации метапеременной STD.

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

- val(e, MeM..STD)=lv(MD(MeM(e)), MeM(e)), если e — обычная переменная;
- val(e, MeM..STD)=e, если e — константа или переменная, к значению которой нет обращения через указатели;
- val(e[e'], MeM..STD)=lv(MD(mb(e, val(val(e', MeM..STD)))), mb(e, val(val(e', MeM..STD))));
- val(e[e'], MeM..STD)=e[val(val(e', MeM..STD))], если e — массив, к значениям элементов которого нет обращения через указатели;
- val(e.m, MeM..STD)=lv(MD(mb(e, m)), mb(e, m));
- val(e.m, MeM..STD)=e.m, если e — структура, к значениям полей которой нет обращения через указатели;
- val(&e, MeM..STD)=loc(val(e, MeM..STD));
- val(*e, MeM..STD)=lv(MD(val(val(e, MeM..STD))), val(val(e, MeM..STD)));
- val((τ) e , MeM..STD)=cast(val(e, MeM..STD), type(e, MeM, TP), fst(logtype(τ, MeM..STD)));
- val(•e, MeM..STD)=UnOpSem(•, val(val(e, MeM..STD)), type(e, MeM, TP)), где • — унарная операция;

- val(e•e', MeM..STD)=BinOpSem(•, val(val(e, MeM..STD)), type(e, MeM, TP), val(val(e', MeM..STD)), type(e', MeM, TP)), где • — бинарная операция.

Пусть τ_1 — несоставной тип, τ_2 — составной тип. *Функция init(τ, e, MeM..TP)* инициализирует содержимое ячейки типа τ в соответствии со спецификатором инициализации e и значениями метапеременных MeM, MD и TP, модифицируя метапеременные MD и TP. Спецификатор инициализации e может быть либо инициализатором, либо классом памяти (если нет инициализатора). В последнем случае инициализация выполняется по умолчанию. Функция init определяется следующим образом:

- init(τ_1, storage, MeM..TP)={(MD, TP, (defaultValue(τ_1, storage), τ_1))};
- init(τ_1 (v, τ'), MeM..TP)={(MD, TP, (cast(v, τ', τ_1), τ_1))};
- init(τ_1, e, MeM..TP)={(MD, TP, (cast(val(e, MeM..STD), type(e, MeM..TP), τ_1), τ_1))};
- init(τ_2, e, MeM..TP)=union(nc: MD(nc)=⊥, union(MD' ∈ updv(upd(MD, nc, ω), nc, τ_2), {(MD", TP", (nc, pointer(TypeOfNewVal(τ)))) | (MD", TP") ∈ updv(MeM, MB', updv(upd(TP, nc, TypeOfNewVal(τ_2)), nc, τ_2), nc, τ_2, e)}.

Функция init использует *функции семейства updv(..., τ)*. Это семейство модифицирует метапеременные MD и TP, чтобы обеспечить размещение объекта составного типа τ. Оно включает три функции.

Функция updv(MD, nc, τ) модифицирует метапеременную MD. Модификация устанавливает значения выделенных ячеек в ω, тем самым сигнализируя, что эти ячейки выделены:

- updv(MD, nc, τ)=updv(MD, nc, τ, findex(τ));
- updv(MD, nc, τ, 0)=updv(MD, nc, τ, next(τ, 0)), если next(τ, 0)≠ω;
- updv(MD, nc, τ, l)=union(nc': MD(nc')≠⊥, updv(upd(MD, nc', ω), nc, τ, next(τ, l), если next(τ, l)≠ω и l≠0;
- updv(MD, nc, τ, l)={MD}, если next(τ, l)=ω.

Функция updv(TP, nc, τ) модифицирует метапеременную TP. Модификация специфицирует типы выделенных ячеек:

- updv(TP, nc, τ)=updv(TP, nc, τ, findex(τ));
- udpv(TP, nc, τ, l)=updv(upd(TP, mb(nc, l), itype(τ, l)), nc, τ, next(l)), если next(τ, l)≠ω;
- updv(TP, nc, τ, l)=TP, если если next(τ, l)=ω.

Функция updv(MeM..TP, nc, τ) модифицирует метапеременные MeM, MD и TP. Модификация MD специфицирует значения выделенных ячеек.

Модификация MD и TP определяет объекты вложенных составных типов (если такие имеются) через вызов функции init. Таким образом, функции updv и init взаимно-рекурсивны:

- updv(MeM..TP, nc, τ, e)=updv(MeM..TP, nc, τ, storage, findex(τ));
- updv(MeM..TP, nc, τ, storage, l)=union((MD', TP', Val') ∈ init(TP(mb(nc, l)), storage, MeM, MD, TP), updv(MeM, upd(MD', fst(Val')), TP', nc, τ, storage, next(l)), если next(τ, l)≠ω;
- updv(MeM..TP, nc, τ, {e_1, …, e_k}, l)=union((MD', TP', Val') ∈ init(TP(mb(nc, l)), e_1, MeM, MD, TP), updv(MeM, upd(MD', fst(Val')), TP', nc, τ, {e_2, …, e_k}, next(l)), если next(τ, l)≠ω;
- updv(MeM..TP, nc, τ, {}, l)=union((MD', TP', Val') ∈ init(TP(mb(nc, l)), static, MeM, MD, TP), updv(MeM, upd(MD', fst(Val')), TP', nc, τ, {}, next(l)), если next(τ, l)≠ω;
- updv(MeM..TP, nc, τ, e, l)={(MD, TP)}, если next(τ, l)=ω.

Пусть τ_1 — несоставной тип, τ_2 — составной тип. *Функция new(τ, MeM..TP)* распределяет память для объекта типа τ, модифицируя метапеременные MD и TP, возвращая указатель на выделенную память, при этом указатель возвращается как пара — адрес и тип указателя:

- new(τ_1, MeM..TP)={(upd(MD, nc, ω), upd(TP, nc, τ_1), (nc, pointer(τ_1)) | MD(nc)=⊥};
- new(τ_2, MeM..TP)=init(τ_2, auto, MeM, MD, TP).
- *Функция delete(MD, TP, nc)* освобождает память, начиная с ячейки nc:
- delete(MD, TP, nc)=delete(MD, TP, nc, τ', findex(τ')), если mb(nc, 0)=nc ∧ ∀i (0≤i ∧ i≤k-1 → mb(nc, i)≠⊥ ∧ mb(nc, k)=⊥ ∧ τ'=array(TP(nc), k);
- delete(MD, TP, nc)=delete(upd(MD, nc, ⊥), upd(TP, nc, ⊥), nc, TP(nc), findex(TP(nc))), если TP(nc) — структура;
- delete(MD, TP, nc)=(MD, TP), если ∀i mb(nc, i)=⊥.
- delete(MD, TP, nc, τ, l)=delete(upd(MD', mb(nc, l), ⊥), upd(TP', mb(nc, l), ⊥), nc, τ, next(τ, l)), если next(τ, l)≠ω ∧ (MD', TP')=delete(MD, TP, mb(nc, l));
- delete(MD, TP, nc, τ, l)=(MD, TP), если next(τ, l)=ω.

3. СМЕШАННАЯ АКСИОМАТИЧЕСКАЯ СЕМАНТИКА MMHSC ЯЗЫКА C-KERNEL

При определении смешанной аксиоматической семантики MMHSC (Mixed Modified Hoare Semantics for C-kernel) для 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, S — 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=ω. 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(MeM..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(MeM..STD) называется постусловием функции f.

Спецификация меток SP_{lab} — это отображение из меток в утверждения. Утверждение SP_{lab}(L) называется инвариантом метки L.

Распространим спецификации пред- и постусловий функций на адреса функций:

1. $SP_{pre}(nc)=SP_{pre}(f)$, если MeM(f)=nc;
2. $SP_{post}(nc)=SP_{post}(f)$, если MeM(f)=nc.

Формула Ψ языка аннотаций может содержать спецификацию предусловий функции SP_{pre} и спецификацию постусловий функции SP_{post}. Пусть $SP_{fun}|=\Psi$ обозначает истинность формулы Ψ 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, Q, R обозначают аннотации, A и S — последовательности операторов, окружение E имеет вид (f, τ, B, cur, ω), а идентификатор всей программы равен Prgm.

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

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

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).

3.1. Правила для выражений

Вызов функции. Правило вывода для функции, обозначаемой выражением e_0, в случае, когда она возвращает значение, имеет вид:

$$\frac{\begin{array}{c}E, SP \vdash \{\exists Val'\ \alpha \wedge P'(x_1 \leftarrow cast(val(val(e_1, MeM..STD)), type(e_1, \\ MeM..STD), \tau_1) \ldots x_n \leftarrow cast(val(val(e_n, MeM..STD)), type(e_n, MeM..STD), \\ \tau_n) \Rightarrow Q'(x_1 \leftarrow cast(val(val(e_1, MeM..STD)), type(e_1, MeM..STD), \tau_1) \ldots x_n \leftarrow \\ cast(val(val(e_n, MeM..STD)), type(e_n, MeM..STD), \tau_n)\}\ A;\ \{Q\}\end{array}}{E, SP \vdash \{P\}\ \mathbf{e=e_0(e_1, \ldots, e_n);}\ A;\ \{Q\}} \quad (1)$$

SP_{fun} — спецификация функции e_0;

$P'=fst(SP_{fun}(val(e_0, MeM..STD)))(MeM..Val)$ — предусловие функции e_0;

$Q'=snd(SP_{fun}(val(e_0, MeM..STD)))(MeM..Val)$ — постусловие функции e_0;

$x_1, \ldots, x_n$ — формальные параметры функции e_0;

$\alpha=\exists MD'\ P(MD \leftarrow MD')(Val \leftarrow Val') \wedge MD=upd(MD', loc(val(e, MeM, MD', TP, STD), cast(fst(Val), snd(Val), type(e, MeM, TP)))$, если e — переменная обычного вида;

$\alpha=\exists e'\ P(e \leftarrow e')(Val \leftarrow Val') \wedge e=cast(fst(Val), snd(Val), type(e, MeM, TP))$, если e — простая переменная паскалевского вида;

$\alpha=\exists v'\ P(v \leftarrow v')(Val \leftarrow Val') \wedge v=upd(v', val(val(i, MeM..STD)), cast(fst(Val), snd(Val), type(v', MeM, TP))$, если $e=v[i]$ — массив паскалевского вида;

$\alpha=\exists s'\ P(s \leftarrow s')(Val \leftarrow Val') \wedge s=upd(s', t, cast(fst(Val), snd(Val), type(s', MeM, TP))$, если $e=s.t$ — структура паскалевского вида.

MD', e', v', s' — новые переменные соответствующих типов.

Данное правило осуществляет подстановку формальных параметров в спецификацию функции e_0 и далее действует аналогично правилу для простого присваивания (см. ниже).

Правило вывода для функции, обозначаемой выражением e_0, в случае, когда она не возвращает значение, имеет вид:

$$\frac{E, SP \vdash \{P \wedge P'(x_1 \leftarrow cast(val(val(e_1, MeM..STD)), type(e_1, MeM..STD), \tau_1) \ldots x_n \leftarrow cast(val(val(e_n, MeM..STD)), type(e_n, MeM..STD), \tau_n) \Rightarrow Q'(x_1 \leftarrow cast(val(val(e_1, MeM..STD)), type(e_1, MeM..STD), \tau_1) \ldots x_n \leftarrow cast(val(val(e_n, MeM..STD)), type(e_n, MeM..STD), \tau_n)\}\ A; \{Q\}}{E, SP \vdash \{P\}\ \mathbf{e_0(e_1, \ldots, e_n);}\ A; \{Q\}} \quad (2)$$

Отличие состоит в том, что не нужно применять подстановки для присваивания, поскольку оно отсутствует.

Операция new. Правило вывода для операции new имеет вид:

$$\frac{E, SP \vdash \{\exists MeM'\ \exists MD'\ \exists TP'\ \exists STD'\ \exists MeM''\ \exists MD''\ \exists TP''\ \exists STD''\ \exists \tau\ \exists V\ P(MeM \leftarrow MeM')(MD \leftarrow MD')(TP \leftarrow TP')(STD \leftarrow STD') \wedge (\tau, STD'') \in logtype(e', MeM..STD) \wedge (MD'', MB'', TP'', V) \in new(\tau, MeM, MD, MB, TP) \wedge MD=upd(MD'', loc(val(e, MeM''..STD'')), cast(fst(V), snd(V), type(e, MeM'', TP''))) \wedge TP=TP'' \wedge STD=STD''\}\ A; \{Q\}}{E, SP \vdash \{P\}\ \mathbf{e=new\ e';}\ A; \{Q\}} \quad (3)$$

В этом правиле происходит вызов абстрактной функции new, которая выделяет память под объект соответствующего типа.

Операция delete. Правило вывода для операции delete имеет вид:

$$\frac{E, SP \vdash \{\exists MD'\ \exists TP'\ \exists MD''\ \exists TP''\ P(MD \leftarrow MD')(TP \leftarrow TP') \wedge (MD'', TP'')=delete(MD', TP', val(val(e, MeM, MD', TP', STD)) \wedge MD=MD'' \wedge TP=TP''\}\ A\ \{Q\}}{E, SP \vdash \{P\}\ \textbf{delete e;}\ A;\ \{Q\}} \quad (4)$$

Функция delete осуществляет освобождение памяти, на которую ссылается указатель e.

Операция присваивания. Пусть выражение e_0 не содержит вызовов функций, операторов new и операторов приведения. Правило вывода для операции присваивания имеет вид:

$$\frac{E, SP \vdash \{\exists MD'\ P(MD \leftarrow MD') \wedge (MD=upd(MD', loc(val(e, MeM, MD', TP, STD)), cast(val(val(e_0, MeM, MD', TP, STD)), type(e_0, MeM, TP), type(e, MeM, TP))))\}\ A;\ \{Q\}}{E, SP \vdash \{P\}\ \textbf{e=e}_0\textbf{;}\ A;\ \{Q\}} \quad (5)$$

где e не является переменной паскалевского вида.

Правило представляет собой модификацию правила классической системы Хоара: подстановки осуществляются на метапеременных, и осуществляется приведение типа выражения e_0 к типу e.

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

$$\frac{E, SP \vdash \{\exists e'\ P(e \leftarrow e') \wedge (e=cast(val(val(e_0(e \leftarrow e'), MeM, MD, TP, STD)), type(e_0(e \leftarrow e'), MeM, TP), type(e, MeM, TP)))\}\ A;\ \{Q\}}{E, SP \vdash \{P\}\ \textbf{e=e}_0\textbf{;}\ A;\ \{Q\}} \quad (6)$$

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

$$\frac{E, SP \vdash \{\exists v'\ P(v \leftarrow v') \wedge (v = upd(v', val(val(i, MeM..STD)), cast(val(val(e_0(v \leftarrow v'), MeM, MD, TP, STD)), type(e_0(v \leftarrow v'), MeM, TP), type(v[i], MeM, TP)))\}\ A;\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{v[i]=e_0;}\ A;\ \{Q\}} \quad (7)$$

Если e=s.t — структура паскалевского вида, то

$$\frac{E, SP \vdash \{\exists s'\ P(s \leftarrow s') \wedge (s = upd(s', t, cast(val(val(e_0(s \leftarrow s'), MeM, MD, TP, STD)), type(e_0(s \leftarrow s'), MeM, TP), type(s.t, MeM, TP)))\}\ A;\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{s.t=e_0;}\ A;\ \{Q\}} \quad (8)$$

3.2. Правила для деклараций

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

$$\frac{E, SP \vdash \{\exists MeM'\ \exists MD'\ \exists TP'\ \exists STD'\ \exists nc\ \exists \tau\ \exists V\ \exists MD''\ \exists TP''\ \exists STD''\ P(MeM \leftarrow MeM')(MD \leftarrow MD')(TP \leftarrow TP')(STD \leftarrow STD') \wedge (\tau, STD'') \in logtype(tp(e), MeM'..STD') \wedge MD'(nc) = \perp \wedge (MD'', TP'', V) \in init(\tau, storage(e), MeM', upd(MD', nc, \omega), TP') \wedge MeM = upd(MeM', id(e), nc)) \wedge MD = upd(MD'', nc, fst(V)) \wedge TP = upd(TP'', nc, \tau) \wedge STD = STD''\}\ A\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{e;}\ A;\ \{Q\}} \quad (9)$$

Для простой переменной e паскалевского вида правило имеет вид:

$$\frac{E, SP \vdash \{\exists e'\ P(e \leftarrow e') \wedge e = defaultValue(\tau, storage)(e \leftarrow e')\}\ A\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{storage\ \tau\ e;}\ A;\ \{Q\}} \quad (10)$$

Если в декларации объявляется массив v паскалевского вида, то

$$\frac{E, SP \vdash \{\exists v'\ P(v \leftarrow v') \wedge v=((dV, \ldots, dV), \ldots(dV, \ldots, dV))\}\ A\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{storage\ \tau[n_1, \ldots, n_k]\ v;}\ A;\ \{Q\}} \quad (11)$$

где dV=defaultValue(τ, storage)(v ← v').

В случае, когда объявляется структура s паскалевского вида, правило имеет вид:

$$\frac{E, SP \vdash \{\exists s'\ P(s \leftarrow s') \wedge s=(defaultValue(\tau_1, storage)(s \leftarrow s'), \ldots, defaultValue(\tau_n, storage)(s \leftarrow s'))\}\ A\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{storage\ struct\ s\ \{\tau_1\ t_1; \ldots; \tau_n\ t_n;\};}\ A;\ \{Q\}} \quad (12)$$

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

$$\frac{E, SP \vdash \{\exists MeM'\ \exists MD'\ \exists TP'\ \exists STD'\ \exists nc\ \exists \tau\ \exists V\ \exists MD''\ \exists TP''\ \exists STD''\ P(MeM \leftarrow MeM')(MD \leftarrow MD')(TP \leftarrow TP')(STD \leftarrow STD') \wedge (\tau, STD'') \in logtype(tp(e), MeM'..STD') \wedge MD'(nc)=\perp \wedge (MD'', TP'', V) \in init(\tau, e_0, MeM', upd(MD', nc, \omega), TP') \wedge MeM=upd(MeM', id(e), nc)) \wedge MD=upd(MD'', nc, fst(V)) \wedge TP=upd(TP'', nc, \tau) \wedge STD=STD''\}\ A\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{e=e_0;}\ A;\ \{Q\}} \quad (13)$$

Для простой переменной e паскалевского вида правило имеет вид:

$$\frac{E, SP \vdash \{\exists e'\ P(e \leftarrow e') \wedge e=e_0(e \leftarrow e')\}\ A\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{storage\ \tau\ e=e_0;}\ A;\ \{Q\}} \quad (14)$$

Если в декларации объявляется массив v паскалевского вида, то

$$\frac{E, SP \vdash \{\exists v'\ P(v \leftarrow v') \wedge v=((v_{0\ldots0}(v \leftarrow v'), \ldots, v_{0\ldots0\ n1\text{-}1}(v \leftarrow v')), \ldots, (v_{nk\text{-}1\ldots nk\text{-}1\ 0}(v \leftarrow v'), \ldots, v_{nk\text{-}1\ldots nk\text{-}1}(v \leftarrow v')))\}\ A\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{storage\ \tau[n_1, \ldots, n_k]\ v=\{\{v_{0\ldots0}, \ldots, v_{0\ldots0\ n1\text{-}1}\}, \ldots, \{v_{nk\text{-}1\ldots nk\text{-}1\ 0}, \ldots, v_{nk\text{-}1\ldots nk\text{-}1}\}\};}\ A;\ \{Q\}} \quad (15)$$

В случае, когда объявляется структура s паскалевского вида, правило имеет вид:

$$\frac{E, SP \vdash \{\exists s'\ P(s \leftarrow s') \wedge s=(v_1(s \leftarrow s'), \ldots, v_n(s \leftarrow s'))\}\ A\ \{Q\}}{E, SP \vdash \{P\}\ \textbf{storage struct s } \{\tau_1\ t_1=v_1; \ldots; \tau_n\ t_n=v_n;\};\ A;\ \{Q\}} \quad (16)$$

Декларации типов. Правило вывода для деклараций типов имеет вид:

$$\frac{E, SP \vdash \{\exists STD'\ \exists \tau\ \exists STD''\ P(STD \leftarrow STD') \wedge (\tau, STD'') \in logtype(e, MeM, MD, TP', STD') \wedge STD=upd(STD'', id(e), \tau)\}\ A;\ \{Q\}}{E, SP \vdash \{P\}\ \textbf{typedef e;}\ A;\ \{Q\}} \quad (17)$$

Декларации функций. Аксиома для декларации функций имеет вид:

$$\frac{E, SP \vdash \{\exists MeM'\ \exists MD'\ \exists TP'\ \exists STD'\ \exists \tau'\ \exists STD''\ \exists nc\ P(MeM \leftarrow MeM')(MD \leftarrow MD')(TP \leftarrow TP')(STD \leftarrow STD') \wedge (\tau', STD'') \in logtype(\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n), MeM'..STD') \wedge MD'(nc)=\perp \wedge MeM=upd(MeM', f, nc) \wedge MD=upd(MD', nc, ([x_1, \ldots, x_n], S)) \wedge TP=upd(TP', nc, \tau') \wedge STD=STD''\}\ A;\ \{Q\}}{E, SP \vdash \{P\}\ \boldsymbol{\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\{S\}}\ A;\ \{Q\}} \quad (18)$$

3.3. Правила для операторов

Помеченный оператор. Правила для помеченного оператора имеют вид:

$$\frac{(f, \tau, B, cur, L), SP \vdash \{P\}\ A;\ \{Q\} \quad (f, \tau, B, cur, \omega), SP \vdash \{SP_{lab}(L_1)\}\ S;\ A;\ \{Q\}}{(f, \tau, B, cur, L), SP \vdash \{P\}\{SP_{lab}(L_1)\}\ L_1: S;\ A;\ \{Q\}}\ L \neq L_1 \quad (19)$$

$$\frac{\begin{array}{c}SP_{fun} \models P \Rightarrow SP_{lab}(L) \\ (f, \tau, B, cur, \omega), SP \vdash \{SP_{lab}(L)\}\ S; A; \{Q\}\end{array}}{(f, \tau, B, cur, E_5), SP \vdash \{P\}\{SP_{lab}(L)\}\ L: S; A; \{Q\}} \quad \begin{array}{l} E_5=\omega \text{ или} \\ E_5=L \end{array} \qquad (20)$$

Правило (19) 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 (20) 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:

$$\frac{(f, \tau, B, id, \omega), SP \vdash \{P\}\ S; blockEnd(cur); A; \{Q\}}{(f, \tau, B, cur, \omega), SP \vdash \{P\}\{S\}_{id}; A; \{Q\}} \qquad (21)$$

$$\frac{(f, \tau, B, cur, \omega), SP \vdash \{\exists MeM'\ P(MeM \leftarrow MeM') \wedge MeM=upd(MeM', blockvars(id), \omega))\}\ A; \{Q\}}{(f, \tau, B, id, \omega), SP \vdash \{P\}\ blockEnd(cur); A; \{Q\}} \qquad (22)$$

$$\frac{(f, \tau, B, cur, L), SP \vdash \{\exists MeM'\ P(MeM \leftarrow MeM') \wedge P(MeM \leftarrow upd(MeM, blockvars(id), \omega))\}\ A; \{Q\}}{(f, \tau, B, id, L), SP \vdash \{P\}\ blockEnd(cur); A; \{Q\}} \qquad (23)$$

$$\frac{(f, \tau, B, cur, id(f)), SP \vdash \{\exists MeM'\ P(MeM \leftarrow MeM') \wedge P(MeM \leftarrow upd(MeM, blockvars(id), \omega))\}\ A;\ \{Q\}}{(f, \tau, B, id, id(f)), SP \vdash \{P\}\ blockEnd(cur);\ A;\ \{Q\}}\ id \neq id(f) \quad (24)$$

$$\frac{SP_{fun} \models \exists MeM'\ P(MeM \leftarrow MeM') \Rightarrow SP_{post}(f) \wedge (MeM = upd(MeM', blockvars(id(f)), \omega))}{(f, \tau, B, id(f), id(f)), SP \vdash \{P\}\ blockEnd(cur);\ A;\ \{Q\}} \quad (25)$$

Правило (21) раскрывает скобки блока, помечая его конец специальной конструкцией blockEnd, которая элиминируется правилами (22) — (25).

Правило (22) соответствует случаю обычного выхода из блока. При этом освобождается память, занятая локальными переменными этого блока.

Правило (23) обеспечивает просачивание метки L: это значит, что ранее был встречен оператор goto L, но метка L внутри блока не найдена.

Правило (24) соответствует случаю, когда был встречен оператор return, но конец блока, которого мы достигли, не является выходом из тела функции.

В случае правила (25) 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:

$$\frac{E, SP \vdash \{P\}\ A;\ \{Q\}}{E, SP \vdash \{P\}\ \mathbf{;}\ A;\ \{Q\}} \quad (26)$$

$$\frac{SP_{fun} \models P \Rightarrow Q}{(f, \tau, B, cur, \omega), SP \vdash \{P\}\ \mathbf{;}\ \{Q\}} \quad (27)$$

$$\frac{SP_{fun} \models P \Rightarrow SP_{lab}(L)}{(f, \tau, B, cur, L), SP \vdash \{P\}\ \mathbf{;}\ \{Q\}} \quad (28)$$

Правило (26) применяется в том случае, когда оставшаяся часть программы A непустая.

Правило (27) соответствует случаю, когда мы достигли конца обрабатываемой части программы. Порождается условие корректности $P \Rightarrow Q$.

В случае правила (28) 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:

$$\frac{\begin{array}{c}E, SP \vdash \{P \wedge cast(val(val(e, MeM..STD)), type(e, MeM, TP), int)\neq 0\}\ S_1; \\ A; \{Q\} \\ E, SP \vdash \{P \wedge cast(val(val(e, MeM..STD)), type(e, MeM, TP), int)=0\}\ S_2; \\ A;\{Q\}\end{array}}{E, SP \vdash \{P\}\ \textbf{if (e) } S_1 \textbf{ else } S_2\textbf{;}\ A;\ \{Q\}} \quad (29)$$

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:

$$\frac{\begin{array}{c}SP_{fun} \models P \Rightarrow INV \\ E, SP \vdash \{INV \wedge cast(val(val(e, MeM..STD)), type(e, MeM,TP), int)\neq 0\} \\ S; \{INV\} \\ E, SP \vdash \{INV \wedge cast(val(val(e, MeM..STD)), type(e, MeM,TP), int)=0\} \\ A\ \{Q\}\end{array}}{E, SP \vdash \{P\}\{INV\}\ \textbf{while (e) S;}\ A;\ \{Q\}} \quad (30)$$

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 имеет вид:

$$\frac{(f, \tau, B, cur, L), SP \vdash \{P\}\ A;\ \{Q\}}{(f, \tau, B, cur, \omega), SP \vdash \{P\}\ goto\ L;\ A;\ \{Q\}} \quad (31)$$

$$\frac{(f, \tau, B, cur, L), SP \vdash \{P\}\ A;\ \{Q\}}{(f, \tau, B, cur, L), SP \vdash \{P\}\ T;\ A;\ \{Q\}} \quad (32)$$

где T не является помеченным оператором и не является blockEnd(cur).

Правило (31) добавляет в окружение метку L с целью поиска соответствующего помеченного оператора, а правило (32) 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 два правила вывода: для return с аргументом и return без аргументов.

$$\frac{(f, \tau, B, cur, id(f)), SP \vdash \{P\}\ A\ \{Q\}}{(f, \tau, B, cur, \omega), SP \vdash \{P\}\ return;\ A\ \{Q\}} \quad (33)$$

$$\frac{(f, \tau, B, cur, id(f)), SP \vdash \{P\}\ A\ \{Q\}}{(f, \tau, B, cur, id(f)), SP \vdash \{P\}\ T;\ A\ \{Q\}} \quad (34)$$

$$\frac{(f, \tau, B, cur, id(f)), SP \vdash \{\exists Val'\ P(Val \leftarrow Val') \wedge Val=(cast(val(val(e, MeM..STD)), type(e, MeM, TP), \tau),\tau)\}\ A\ \{Q\}}{(f, \tau, B, cur, \omega), SP \vdash \{P\}\ return\ e;\ A\ \{Q\}} \quad (35)$$

где T не является помеченным оператором и не является blockEnd(…).

Правила (33) и (34) аналогичны правилам (31) и (32) 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 (35) соответствует случаю, когда функция возвращает значение. В этом случае результат сохраняется в метапеременной Val.

3.4. Другие правила

Правило консеквенции. Правило консеквенции (усиления предусловия и/или ослабления постусловия) имеет вид:

$$\frac{SP_{fun}|= P \Rightarrow R \qquad E, SP \vdash \{R\}\ S\ \{T\} \qquad SP_{fun}|= T \Rightarrow Q}{E, SP \vdash \{P\}\ S\ \{Q\}} \tag{36}$$

Последовательность операторов. Пусть T и T' — непустые последовательности операторов и вспомогательных конструкций. Семантика последовательности операторов характеризуется тем фактом, что свойства в неизменной форме могут переноситься из постусловия предшествующего оператора в предусловие последующего:

$$\frac{E, SP \vdash \{P\}\ T\ \{R\} \qquad E, SP \vdash \{R\}\ T'\ \{Q\}}{E, SP \vdash \{P\}\ T\ T'\ \{Q\}} \tag{37}$$

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

$$\frac{\begin{array}{c}(f, \tau_f, S_f, bid_f, \omega), ((SP_{pre}, SP_{post}), SP^{f}_{lab}) \vdash \{\exists MeM'\ SP_{pre}(f)(x_1, \ldots, x_n)\ (MeM..Val)(MeM \leftarrow MeM') \wedge (MeM=upd(MeM', x_1, nc_1) \wedge \ldots \wedge \\ (MeM=upd(MeM', x_n, nc_n)\}\ S_f; blockEnd(bid_f)\ \{SP_{post}(f)(x_1, \ldots, x_n)\ (MeM..Val)\}\ \text{по всем именам функций f, определённых в T, кроме main,} \\ (main, \tau_{main}, S_{main}, bid_{main}, \omega), SP \vdash \{P\}\ T\ S_{main}; blockEnd(bid_{main})\ \{Q\}\end{array}}{E, SP \vdash \{P\}\ Prgm(T)\ \{Q\}} \tag{38}$$

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

4. ПРИМЕРЫ

Рассмотрим несколько примеров, иллюстрирующих применение правил вывода смешанной аксиоматической семантики.

4.1. Вычисление факториала

Первый пример тривиален, но важен тем, что при выводе применяются правила специального вида (для паскалевских переменных), что позволяет получать довольно простые условия корректности.

Исходная программа на языке C-light имеет вид (будем верифицировать только саму функцию вычисления факториала):

```
/ % n≥0 % /
int factorial(int n)
{
  int i, P=1;
  / % P=(i-1)! ∧ i≤n+1 % /
  for (i=1;i<=n;i++) P*=i;
  return P;
}
/ % Val=(n!, int) % /
```

Соответствующая ей программа на языке C-kernel выглядит так:

```
/ % n≥0 % /
int factorial(int n)
{
  auto int i;
  auto int P=1;
  {
        i=1;
        / % P=(i-1)! ∧ i≤n+1 % /
                while (i<=n)
        {
              P=P*i;
              i=i+1;
        }body
  }for
  return P;
```

```
}factorial
/% Val=(n!, int) %/
```

В этой программе нет обращений ни к одному значению ни одной переменной через указатели (т. е. они все имеют паскалевский вид) и имеется единственный цикл. Поэтому в результате вывода получатся три условия корректности:

1. $\exists i''\ (\exists P\ (\exists i'\ (n\geq 0 \wedge i''=0) \wedge P=1) \wedge i=1) \Rightarrow P=(i-1)! \wedge i\leq n+1$

Доказательство очевидно, так как 0!=1, а $1\leq n+1$ в силу $n\geq 0$.

2. $\exists i'\ (\exists P'\ (P'=(i'-1)! \wedge i'\leq n+1 \wedge i'\leq n \wedge P=P'*i') \wedge i=i'+1) \Rightarrow P=(i-1)! \wedge i\leq n+1$

Доказательство: $P=P'*i'=(i'-1)!*i'=i'!=(i-1)!$

$i'\leq n \Rightarrow i'+1\leq n+1 \Rightarrow i\leq n+1$.

3. $P=(i-1)! \wedge i\leq n+1 \wedge \neg(i\leq n) \wedge Val=(P, int) \Rightarrow Val=(n!, int)$

Доказательство: $n+1\geq i>n \Rightarrow i=n+1 \Rightarrow P=n!$

Тем самым установлена частичная корректность программы вычисления факториала.

4.2. Aliasing

Этот пример иллюстрирует, как с помощью метапеременных можно успешно верифицировать программу, где доступ к одному и тому же участку памяти осуществляется через несколько программных идентификаторов.

Пусть имеется программа:

```
/% MD(MeM(i))=i0 %/
int aliasing(int i)
{
  int* p=&i;
  (*p)++;
  i++;
  return i;
}
/% Val=(i0+2,int) %/
```

Соответствующая ей программа на языке C-kernel:

```
/% MD(MeM(i))=i0 %/
int aliasing(int i)
{
  auto int* p=&i;
```

```
  (*p)=(*p)+1;
  i=i+1;
  return i;
}aliasing
/ % Val=(i0+2,int) % /
```

Поскольку к значению переменной i есть доступ через указатель p, то мы не можем применить специальное правило (6), поэтому применяется правило (5). Получаем единственное условие корректности:

1. $\exists MD'''(\exists MD''(\exists MD'(MD'(MeM(i))=i_0 \wedge MD''=upd(MD', MeM(p), MeM(i))) \wedge MD'''=upd(MD'', MD''(MeM(p)), MD''(MD''(MeM(p)))+1)) \wedge MD=upd(MD''', MeM(i), MD'''(MeM(i))+1) \wedge Val=(MD(MeM(i)), int)) \Rightarrow Val=(i_0+2, int)$

Доказательство состоит в последовательной подстановке равенств:

$Val=(MD(MeM(i)), int)) \Rightarrow Val=(MD'''(MeM(i))+1, int)), MD'''=upd(MD'', MD''(MeM(p)), MD''(MD''(MeM(p)))+1) \Rightarrow Val=(MD'''(MeM(i))+1, int)), MD'''=upd(MD', MeM(i), MD'(MeM(i))+1) \Rightarrow Val=(MD'(MeM(i))+1+1, int)) \Rightarrow Val=(MD'(MeM(i))+2, int)) \Rightarrow MD'(MeM(i))=i_0 \Rightarrow Val=(i_0+2, int))$

4.3. Поиск в односвязном списке

Рассмотрим функцию поиска по заданному ключу элемента односвязного списка, состоящего из целых чисел. Программа имеет вид:

```
struct list
{
  int key;
  struct list* next;
}
/ % pattern=n0 ∧ MD(MeM(source))=s0 % /
struct list* search(struct list* source, int pattern)
{
          / % pattern=n0 ∧ n0∉dellast(tuple(s0, MD(MeM(source)))) ∧
      reach(MD(MeM(source)), s0) % /
  while (source!=NULL)
        if (source->key==pattern) {break;}
else {source=source->next;}
  return source;
}
```

/ % pattern=$n_0 \wedge n_0 \notin$dellast(tuple(s_0, fst(Val))) $\wedge$ reach(MD(MeM(fst(Val))), s_0) $\wedge$ (MD(MeM(source))$\neq$NULL $\Rightarrow$ MD(mb(MD(MeM(fst(Val))), key))=n_0) $\wedge$ (MD(MeM(source))=NULL $\Rightarrow$ MD(mb(MD(MeM(fst(Val))), key))= ω) % /

Соответствующая ей программа на C-kernel имеет вид:

```
struct list
{
  int key;
  struct list* next;
}
/ % pattern=n0 ∧ MD(MeM(source))=s0 % /
struct list* search(struct list* source, int pattern)
{
  {
                / % pattern=n0 ∧ n0∉dellast(tuple(s0,
            MD(MeM(source)))) ∧ reach(MD(MeM(source)), s0) % /
      while (source!=NULL)
      {
          if (source->key==pattern) {goto L;}
                    else {source=source->next;}
      }
                / % pattern=n0 ∧ n0∉dellast(tuple(s0,
            MD(MeM(source)))) ∧ reach(MD(MeM(source)), s0) ∧
            (MD(MeM(source))≠NULL ⇒ MD(mb(MD(MeM(source)),
            key))=n0) ∧ (MD(MeM(source))=NULL ⇒
            MD(mb(MD(MeM(source)), key))=ω) % /
      L:
  }
  return source;
}
/ % pattern=n0 ∧ n0∉dellast(tuple(s0, fst(Val))) ∧
reach(MD(MeM(fst(Val))), s0) ∧ (fst(Val)≠NULL ⇒
MD(mb(MD(MeM(fst(Val))), key))=n0) ∧ (fst(Val)=NULL ⇒
MD(mb(MD(MeM(fst(Val))), key))= ω) % /
```

Функция tuple(x, y) возвращает кортеж значений элементов списка, начиная с элемента x и y.

Функция dellast($(n_1, ..., n_{k-1}, n_k)$) удаляет последний элемент кортежа:

dellast($(n_1, \ldots, n_{k-1}, n_k)$)=$(n_1, \ldots, n_{k-1})$.

Пусть $X=(x_1, \ldots, x_n)$, $Y=(y_1, \ldots, y_k)$, тогда *функция con(X, Y)* осуществляет соединение элементов кортежей X и Y: $con(X, Y)=(x_1, \ldots, x_n, y_1, \ldots, y_k)$.

Из определений этих двух функций немедленно следует свойство:

$tuple(x, y)=con(dellast(tuple(x, y)), tuple(y, y))$.

Предикат reach(y, x) истинен тогда и только тогда, когда элемент списка $y \neq x$ достижим из элемента x, то есть существует конечная последовательность x->next-> … ->next=y, а $reach(x, x)$ — тождественная истина.

Из определения немедленно следует свойство: $reach(y, x) \wedge reach(z, y) \Rightarrow reach(z, x)$.

Получим следующие условия корректности:

1. $pattern=n_0 \wedge MD(MeM(source))=s_0 \Rightarrow pattern=n_0 \wedge n_0 \notin dellast(tuple(s_0, MD(MeM(source)))) \wedge reach(MD(MeM(source)), s_0)$.

Поскольку $MD(MeM(source))=s_0$, то $tuple(s_0, MD(MeM(source)))=tuple(s_0, s_0)$ является одноэлементным кортежем, следовательно, $dellast(tuple(s_0, s_0))=\varnothing$ — пустой кортеж, а $reach(s_0, s_0)$ — истина, поэтому заключение импликации истинно.

2. $pattern=n_0 \wedge n_0 \notin dellast(tuple(s_0, MD(MeM(source)))) \wedge reach(MD(MeM(source)), s_0) \wedge MD(MeM(source)) \neq NULL \wedge MD(mb(MD(MeM(source)), key))=pattern \Rightarrow (pattern=n_0 \wedge n_0 \notin dellast(tuple(s_0, MD(MeM(source)))) \wedge reach(MD(MeM(source)), s_0) \wedge (MD(MeM(source)) \neq NULL \Rightarrow MD(mb(MD(MeM(source)), key))=n_0) \wedge (MD(MeM(source))=NULL \Rightarrow MD(mb(MD(MeM(source)), key))=\omega))$

Так как $pattern=n_0$ и $MD(MeM(source)) \neq NULL$,
а $MD(mb(MD(MeM(source)), key))=pattern$,
то $MD(mb(MD(MeM(source)), key))=n_0$.

3. $\exists MD'\ (pattern=n_0 \wedge n_0 \notin dellast(tuple(s_0, MD'(MeM(source)))) \wedge reach(MD'(MeM(source)), s_0) \wedge MD'(MeM(source)) \neq NULL \wedge MD'(mb(MD'(MeM(source)), key)) \neq pattern) \wedge MD=upd(MD', MeM(source), mb(MD'(MeM(source)), next)) \Rightarrow pattern=n_0 \wedge n_0 \notin dellast(tuple(s_0, MD(MeM(source)))) \wedge reach(MD(MeM(source)), s_0)$

В силу равенства MD=upd(MD', MeM(source), mb(MD'(MeM(source)), next)) в посылке и по определению функции upd получаем, что MD(MeM(source))=mb(MD'(MeM(source)), next).

По определению предиката reach имеем, что

reach(mb(MD'(MeM(source)), next), MD'(MeM(source))) — истина, но $reach(MD'(MeM(source)), s_0)$ — истина, поэтому по свойству предиката получаем истинность

reach(mb(MD'(MeM(source)), next), s_0)=reach(MD(MeM(source)), s_0).

Поскольку pattern=n_0, то MD'(mb(MD'(MeM(source)), key))≠n_0, но tuple(s_0, mb(MD'(MeM(source)), next)) = con(tuple(s_0, MD'(MeM(source))), tuple(mb(MD'(MeM(source)), next), mb(MD'(MeM(source)), next))) следовательно, $n_0 \notin$ dellast(tuple(s_0, mb(MD'(MeM(source)), next))) = dellast(tuple(s_0, MD(MeM(source)))).

4. pattern=$n_0 \wedge n_0 \notin$dellast(tuple(s_0, source)) $\wedge$ reach(MD(MeM(source)), s_0) $\wedge$ MD(MeM(source))=NULL $\Rightarrow$ (pattern=$n_0 \wedge n_0 \notin$dellast(tuple(s_0, source)) $\wedge$ reach(MD(MeM(source)), s_0) $\wedge$ (MD(MeM(source))≠NULL $\Rightarrow$ MD(mb(MD(MeM(source)), key))=n_0) $\wedge$ (MD(MeM(source)) = NULL $\Rightarrow$ MD(mb(MD(MeM(source)), key))=ω))

Истинность данного условия очевидна, т. к. в силу MD(MeM(source)) = NULL значение поля key не определено.

5. $\exists$Val' (pattern=$n_0 \wedge n_0 \notin$dellast(tuple(s_0, MD(MeM(source))) $\wedge$ reach(MD(MeM(source)), s_0) $\wedge$ (MD(MeM(source))≠NULL $\Rightarrow$ MD(mb(MD(MeM(source)), key))=n_0) $\wedge$ (MD(MeM(source))=NULL $\Rightarrow$ MD(mb(MD(MeM(source)), key))=ω)) $\wedge$ Val=(MD(MeM(source)), struct list *) $\Rightarrow$ pattern=$n_0 \wedge n_0 \notin$dellast(tuple(s_0, fst(Val))) $\wedge$ reach(fst(Val), s_0) $\wedge$ (fst(Val)≠NULL $\Rightarrow$ MD(mb(fst(Val), key))=n_0) $\wedge$ (fst(Val)=NULL $\Rightarrow$ MD(mb(fst(Val), key))=ω)

В силу того, что fst(Val)=MD(MeM(source)), данное условие корректности истинно. Таким образом, программа поиска в линейном односвязном списке частично корректна.

5. ЗАКЛЮЧЕНИЕ

Данная работа продолжает цикл статей, связанных с проектом лаборатории теоретического программирования ИСИ СО РАН по верификации программ, написанных на языке C-light. Предложенная модифицированная смешанная аксиоматическая семантика MMHSC является основой реализуемого в настоящий момент генератора условий корректности автоматизированной системы верификации C-light программ. Однако в работе отсутствует обоснование корректности правил вывода относительно операционной семантики языка C-light. Такое обоснование предполагается выполнить в дальнейшем.

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: Изд-во СО РАН, 2004. — Вып. 9: Формальные методы и модели информатики. — С. 51–134.
2. Непомнящий В. А., Рякин, О. М. Прикладные методы верификации программ. — М.: Радио и связь, 1988.
3. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. — 1969. — Vol. 12, № 1. — P. 576–580.
4. Непомнящий В. А. Символический метод верификации финитных итераций над изменяемыми структурами данных // Программирование. — 2005. — Т. 31, № 1. — С. 1–9.
5. Марьясов И. В. Автоматическая верификация программ на языке C-light // VIII Всероссийская конференция молодых учёных по математическому моделированию и информационным технологиям: Программа и тезисы докладов. — Новосибирск: Институт вычислительных технологий СО РАН, 2007. — С. 103.
6. Марьясов И. В. Автоматическая верификация программ на языке C-light // Технологии Microsoft в теории и практике программирования: конференция-конкурс работ студентов, аспирантов и молодых учёных: тезисы докладов. — Новосибирск: Новосибирский государственный университет, 2007. — С. 25–27.

И. В. Марьясов

НА ПУТИ К АВТОМАТИЧЕСКОЙ ВЕРИФИКАЦИИ ПРОГРАММ НА ЯЗЫКЕ C-LIGHT. СМЕШАННАЯ АКСИОМАТИЧЕСКАЯ СЕМАНТИКА ЯЗЫКА C-KERNEL

Препринт
150

Рукопись поступила в редакцию 20.12.08
Редактор Т. М. Бульонкова
Рецензент И. С. Ануреев

Подписано в печать 27.12.08
Формат бумаги 60 × 84 1/16 Объем 1.8 уч.-изд.л., 2.0 п.л.
Тираж 60 экз.

Центр оперативной печати «Оригинал 2»
г.Бердск, ул. О. Кошевого, 6, оф. 2, тел. (383-41) 2-12-42