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

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

НА ПУТИ К ВЕРИФИКАЦИИ С-ПРОГРАММ. ЧАСТЬ 2. ЯЗЫК C-LIGHT-KERNEL И ЕГО АКСИОМАТИЧЕСКАЯ СЕМАНТИКА

Препринт
87

Новосибирск 2001

Описан язык C-light-kernel, являющийся удобным для верификации ядром языка C-light. Определена аксиоматическая семантика языка C-light-kernel и доказана ее непротиворечивость относительно операционной. Предполагается использование языка C-light в качестве входного языка системы верификации программ, которая включает транслятор в C-light-kernel, а также генератор условий корректности C-light-kernel-программ, базирующейся на его аксиоматической семантике.

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

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

TOWARDS THE VERIFICATION OF C PROGRAMS. PART 2. LANGUAGE C-LIGHT-KERNEL AND ITS AXIOMATIC SEMANTICS

Preprint
87

Novosibirsk 2001

A language C-light-kernel suitable for verification is described. This language is a kernel of the C-light language. Axiomatic semantics of C-light-kernel is defined and its soundness with respect to operational semantics is proved. We suggest to use C-light as an input language for a program verification tool which includes a translator to C-light-kernel and a verification condition generator for C-light-kernel programs.

1. ВВЕДЕНИЕ

Формальная верификация программ — актуальное направление современного программирования. Особый интерес представляет верификация программ, написанных на распространенных языках системного программирования таких, как C и C++. Трудности верификации программ на языке C подробно обсуждались в [2, 6–8, 13, 14]. Напомним, что основной проблемой является отсутствие формальной семантики для полного языка C, соответствующего стандарту ANSI. Некоторые низкоуровневые возможности C также могут стать причиной семантических трудностей. Первым шагом на пути к верификации C-программ стало создание языка C-light [2], являющегося представительным подмножеством языка ANSI C, расширенного некоторыми элементами языка C++. Сравнительный анализ работ, посвященных анализу и верификации C-программ, показал, что это одно из наиболее обширных подмножеств языка C. Оно содержит все множество операторов языка C при некоторых семантических ограничениях и большинство типов и операций. C-light имеет детерминированную семантику выражений, допускает только ограниченное использование операторов switch и goto, а вместо библиотечных функций включает операции new и delete языка C++ для работы с динамической памятью. C-light не поддерживает поразрядных операций, типа объединения, структур с битовыми полями, ряда модификаторов и спецификаторов типов, функций с переменным числом аргументов.

Создание формальной семантики является необходимым условием для верификации программ на языке C-light. Для этой цели был выбран метод структурной операционной семантики [15]. Полное описание семантики языка C-light было приведено в [2]. Там же описаны формальный синтаксис и ограничения.

Верификация программ на основе операционной семантики приводит к громоздким доказательствам [14]. Лучшим вариантом является применение аксиоматической семантики, предложенной Хоаром. Это семантика более высокого уровня, что позволяет существенно упростить доказательство корректности программ. Однако аксиоматическая семантика некоторых конструкций языка C-light является весьма громоздкой, что усложняет верификацию C-light-программ. Выходом стало применение двухуровневой схемы верификации C-light программ. В соответствии с этой схемой в языке C-light выделяется ядро, «хорошее» с точки зрения аксиоматической семантики, в которое транслируются

исходные программы. Цель данной работы — описание и обоснование аксиоматической семантики этого ядра, названного C-light-kernel.

В разд. 2 рассмотрены язык C-light-kernel и язык спецификации программ. Формальный синтаксис языка C-light-kernel не приводится, поскольку ряд ограничений на язык C-light, в соответствии с которыми происходит перевод, носит контекстный характер. Описание задано на основе формального синтаксиса языка C-light [2], поэтому перечислены лишь особенности языка, имеющие непосредственное отношение к верификации, а также ограничения. Язык спецификации программ — это язык логических утверждений о свойствах программ. Он используется при описании состояний абстрактной машины и для задания аннотаций в аксиоматической семантике.

Разд. 3 посвящен формальному определению языка C-light-kernel в виде структурной операционной семантики (СОС). На основе понятия состояния абстрактной вычислительной машины и интерпретации выражений языка спецификации формулируются правила перехода между конфигурациями абстрактной машины. При этом семантика языка C-light-kernel существенно проще семантики языка C-light. Основное определение корректности программы дается в терминах операционной семантики.

В разд. 4 определяется аксиоматическая семантика HSC для языка C-light-kernel. Эта система вывода сводит проблему истинности тройки Хоара для исходной программы к проблеме истинности набора условий корректности. Отдельно рассмотрена аксиоматическая семантика указателей. Доказательство теоремы о непротиворечивости аксиоматической семантики относительно операционной приводится в разд. 5.

Эта работа частично поддержана грантом РФФИ 00-01-00909.

2. ОБЗОР ЯЗЫКА C-LIGHT-KERNEL

Язык аннотированных программ C-light-kernel является ядром языка C-light и, соответственно, подмножеством «чистого» C, расширенным некоторыми лексическими правилами из C++ и аннотациями. Любой текст, соответствующий одновременно синтаксису C-light-kernel и синтаксису ANSI C, будет иметь одинаковый смысл в обоих языках.

Так же как и обычный C, язык C-light-kernel является макроассемблером, хотя и с очень высоким уровнем абстракции. В отличие от других языков высокого уровня, таких как Pascal, ни в C ни в C-light-kernel нет неявно вводимой «виртуальной машины», умеющей манипулиро-

вать такими объектами высокого уровня, как, например символьные строки. Знание этого факта позволяет описывать язык, сопоставляя каждой конструкции C-light-kernel некоторое понятие фон-неймановской вычислительной машины с линейной памятью.

В п. 2.1 приведено неформальное описание языка программ. В п. 2.2 излагается язык описания аннотаций.

2.1. Язык программ

Синтаксис лексем языка C-light-kernel совпадает с синтаксисом лексем языка C-light [2], однако множество ключевых слов сокращается, и остаются следующие: bool, case, char, default, delete, double, else, false, float, goto, if, int, long, new, return, short, signed, sizeof, struct, switch, true, typedef, unsigned, void, wchar_t, while.

Главным отличием языка C-light-kernel от C-light относительно системы типов является то, что все типы, которые допускались синтаксически в C-light, но не поддерживались в семантике, в C-light-kernel запрещены полностью. Кроме того отсутствуют перечисления, так как при переводе они заменяются на декларации констант типа signed int. Все остальные ограничения из языка C-light попадают и в C-light-kernel.

Имена. В C-light-kernel имя объекта — это просто идентификатор. При переводе исчезает операция квалификации :: и все объекты имеют уникальные имена.

Аннотации. В отличие от C, в C-light-kernel введено понятие *аннотации*. Аннотация — это фрагмент текста, обрамленный специальными лексемами-скобками /% и %/. Текст есть правильная аннотация, если он может быть разбит на лексемы и пробельные элементы C-light-kernel без лексических ошибок и в нем соблюден «баланс скобок». Аннотации используются для записи спецификаций программ. Язык спецификаций рассмотрен далее.

Отметим, что аннотации в программе на языке C-light-kernel могут отличаться от исходных аннотаций в C-light-программе, так как при переводе в них могли происходить подстановки.

Среди аннотаций выделяются управляющие, позволяющие определить способ обработки тела функции. Тройки Хоара для inline функций будут доказываться обязательно, тройки extern функций будут считаться аксиомами, тройка auto функции будет доказываться, только если эта функция прямо или косвенно вызывается из одной из inline-функций.

Слова inline, extern и auto — служебные слова аннотации, а не ключевые слова языка программ.

Декларации. Декларации устанавливают соответствие между идентификаторами и объектами в памяти. В C-light-kernel, как и в ANSI C, объекты-данные строго отделены от объектов-тел функций. Объекты-данные могут быть вложены в другие объекты-данные и в общем случае могут изменяться во время работы программы; взаимное расположение этих объектов в памяти может существенно использоваться в реализуемых программой алгоритмах. Тела функций не могут быть вложены друг в друга, не могут изменяться, их размер и порядок в памяти заранее не известен, и над их адресами обычно не производится никаких операций, кроме, разве что, проверки равенства. В связи с этим C-light-kernel строго определяет раскладку в памяти объектов-данных, но не делает никаких утверждений относительно объектов-тел функций.

Как и в ANSI C, декларации подразделяются на *определяющие декларации*, или *определения*, которые помимо сопоставления объекта идентификатору дополнительно определяют время, место и другие параметры создания объекта, и на *ссылочные декларации*, или *прототипы*, которые описывают идентификатор и тип объекта, но не создают сам объект. Объект может иметь несколько не противоречащих друг другу деклараций, но не может иметь несколько определений.

Если объект есть составная часть другого объекта, то для него в тексте программы не может существовать отдельных деклараций, однако его определение может находиться внутри тела декларации объемлющего объекта, например: определение члена структуры может находиться внутри определения всей этой структуры; определение элемента инициализированного массива присутствует в виде значения в списке параметров инициализации этого массива.

В языке C-light-kernel понятие неокончательного определения tentative definition) из языка ANSI C отсутствует.

Принципиальное отличие языка C-light-kernel как от C, так и от C-light, заключается в том, что глобальными объектами могут быть только указатели (и соответственно ссылочные классы, рассмотренные далее). Это ограничение снимает ряд существенных трудностей семантики функций и процедур. При переводе глобальные объекты будут собираться в специальную структуру, а функциям будет передаваться указатель на нее.

В языке C-light-kernel локальное переопределение переменных отсутствует — все различные объекты должны иметь уникальные имена. В частности, при трансляции имена автоматических объектов будут уточняться за счет добавления информации о строке и столбце в исходном файле.

В отличие от языков C и C-light одна декларация не может содержать целый список декларируемых объектов. Таким образом любая декларация объявляет ровно один объект. Случай одновременной декларации структуры и typedef не является исключением, поскольку типы являются объектами периода компиляции, но не периода исполнения.

Объект может быть полностью или частично инициализирован. Составной объект, такой как структура или массив, может быть инициализирован только константными выражениями и только если он является статическим.

После имени объекта могут быть указаны такие модификаторы типа как "массив из элементов типа ..."и "функция, возвращающая значение типа ...". В отличие от C-light, разрешающего, например описание типа многомерного массива непосредственно в декларации (например `int a[2][3]`), C-light-kernel требует использования предварительно созданного именованного типа.

Модификатор "функция, возвращающая значение типа ..."не только описывает тип, но и может назначить имена некоторым объектам — аргументам функции. Запрещены списки аргументов переменной длины. Пустой список аргументов не разрешен — требуется void. Декларация отдельного аргумента функции может быть либо «обычной», либо абстрактной декларацией.

Последнее важное отличие от языка C-light относится к спецификации декларации. Она перечисляет свойства объекта, которые нужны в основном для оптимизации кода. В языке C-light большинство спецификаторов и модификаторов игнорировались, так как операционная среда C-light существенно проще реальной операционной среды C. Но в C-light-kernel спецификаторы класса памяти static и auto становятся обязательными, что позволяет просто задать семантику неявной инициализации.

Примеры деклараций в языке C-light-kernel:

```
int main(int argc, const char *argv[]);
typedef struct cplx{double re, im; } COMPLEX;
int *a, b=0, c[3];              //синтаксическая ошибка
```

```
auto int b_squared = b*b          //Инициализация скаляром
COMPLEX pi = {3.1415, 0}          //Инициализация списком
COMPLEX arr[100] = {{3.1415, 0}, {0, 1}}
                                  //Частичная инициализация

COMPLEX twopi = {2*3.1415, 0 }  //Допустимый список
                                  //инициализаторов
COMPLEX F1 = {calcF(1), 0 }     //Ошибка: не-константное
                                  //выражение

static char c3[] ="abc";
```

Выражения. В языке C-light допускались все выражения языка C, несодержащие низкоуровневых операций. Синтаксис и семантика выражений языка C-light-kernel гораздо проще. Они получаются из выражений языка C-light максимально возможным разбиением на составные части, и эти части записываются как последовательность операторов вычисления выражения.

Допустимыми операциями языка C-light-kernel являются следующие:

1) **первичные**: () [] . ->

2) **унарные**: * - !

3) **бинарные**: * / % + - < > <= >= == !=

4) **присваивание**: =

5) **выделение памяти**: new

6) **освобождение памяти**: delete

7) **приведение типа**.

Следует отметить отсутствие побитовых операций, операции взятия адреса, логических операций И и ИЛИ, операций составного присваивания, условной операции и операции последовательного вычисления. Все эти операции были либо запрещены изначально в языке C-light (побитовые), либо переписывались при трансляции (см. далее).

Выражение присваивания может содержать операции присваивания, а может быть простым rvalue. В отличие от C-light, в C-light-kernel есть ровно одна операция простого присваивания, и отсутствуют операции

наподобие +=. Кроме того, не разрешаются цепочки присваиваний вида a=b=c. В C-light-kernel нет операции «запятая», поэтому любое выражение есть либо выражение присваивания, либо вызов процедуры. Таким образом, выражение присваивания обретает тот же статус, что и оператор присваивания в языке Паскаль.

Одна из целей трансляции из языка C-light в язык C-light-kernel — устранение побочных эффектов. Поэтому фактическим аргументом функции может быть либо константа, либо переменная, но не выражение с какими-либо операциями. Помимо этого важным ограничением является следующее: если указатель ptr есть параметр функции, возвращающей значение, то в теле этой функции выражение вида *ptr не может находиться слева от операции присваивания. На процедуры это ограничение не распространяется. Т. е., если функция возвращает значение, то она не может изменять аргументов, передаваемых через указатели. Такое ограничение обычно является правилом хорошего стиля в программировании, но в нашем случае его необходимо жестко фиксировать, иначе соответствующее правило аксиоматической семантики в общем случае окажется неверным. При трансляции из языка C-light функция, в которой есть такая ситуация, преобразуется в процедуру с тем же именем и дополнительным параметром. Как было сказано выше, при трансляции также контролируется изменение глобальных объектов.

Для приведения типов используется синтаксис языка C. Допустимы только безопасные приведения типов, главным образом для скалярных типов. Для указателей введено дополнительное ограничение — можно делать только приведение от типа void* к T*, где T — любой стандартный тип или тип пользователя. Причина таких ограничений в том, что корректная интерпретация этой операции в семантике требует явно привязываться к конкретной архитектуре (см., например [14]), что усложнит семантику. Кроме того, эта операция определена не для любых типов, а указывать в семантике условия применимости этой операции громоздко. Поэтому рассматриваем только безопасные приведения скалярных типов и указателей (через пустой тип), и семантика приведений определяется семейством функций $\gamma_{T',T}$ (см. п. 3.1).

Для выделения и освобождения динамической памяти используются операции языка C++. Причина состоит в следующем. Операционная семантика описывает вызов функции как подстановку аргументов и переход к телу фунции. Т. е. для обработки стандартных библиотечных функций malloc, calloc и free нужен доступ к их исходным файлам. Да-

лее, эти функции явно изменяют кучу (heap) и достаточно сложны, что может привести к очень громоздким условиям корректности. Выходом является накапливание знаний о средствах работы с памятью. Но выделять отдельно стандартные функции неразумно — это приведет к неоднородной семантике. Предлагается использовать не функции, а операции языка C++, и задавать их семантику более абстрактно (см. п. 4.2).

Формальный синтаксис выражений и их типы (т. е. статическая семантика языка) рассмотрены в п. 2.2.

Операторы. Хотя набор операторов языка C-light-kernel ограничен, он позволяет записать любую программу на языке C-light. Перечислим все операторы.

• Оператор объявление. Декларации данных, используемые для создания переменных, членов структур, функций и т.д., подробно рассмотрены выше.

• Оператор вычисления выражения (возможно пустой), также рассмотрен ранее.

• Условный оператор if.

• Оператор переключатель switch.

• Оператор цикла while.

• Оператор передачи управления goto.

• Составной оператор (блок).

При этом, в сравнении с языком C, есть ряд дополнительных ограничений.

В условном операторе if всегда есть ветвь else, возможно пустая. Каждая ветвь в операторе switch заканчивается оператором goto, передающим управление на оператор, следующий за телом оператора switch. Все метки должны находиться на одном уровне вложенности, т. е. следующий вариант запрещен:

```
switch(i){
    case 1: if(a>0) {case 2: b = 3;}
            else {case 3: c = 0;}}
```

Не предусмотрено никакого особого места для записи инварианта цикла. По соглашению, инвариантом будет считаться первая аннотация в составном операторе — теле цикла.

Запрещено передавать управление по goto внутрь любого блока из охватывающего его блока, или из одного блока в другой, не пересекающийся с ним. Однако можно передать управление из вложенного блока

в охватывающий. Как и в C++ запрещено передавать управление по goto в обход инициализации.

Исходная программа. Весь исходный текст программы есть последовательность внешних деклараций. На внешнем уровне можно объявить тип (typedef-декларации), функцию (прототип или определение), структуру и данные. Кроме обычных конструкций C на верхнем уровне могут присутствовать и некоторые типы аннотаций.

В языке C-light-kernel препроцессор отсутствует, поскольку исходная программа на языке C-light перед трансляцией препроцессируется.

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

2.2. Язык аннотаций

Для записи аннотаций и описания состояний абстрактной машины требуется специальный язык утверждений — язык спецификации программ. Важным требованием к этому языку является возможность записи выражений языка C-light-kernel. Язык спецификаций строится на основе языка исчисления предикатов первого порядка.

2.2.1. Типы и алфавит

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

Здесь T и T_i — любые непустые типы со стандартными ограничениями языка C: массив не может быть параметром или возвращаемым значением функции, функция не может быть параметром другой функции, функции не могут быть элементами массивов.

Базовый набор типов языка C расширен новым параметрическим типом — ссылочным классом $\mathbf{Ref}(T)$. Этот тип необходим для работы с указателями. Ссылочный класс ведет себя как неограниченный массив элементов типа T, роль индексов в котором играют указатели. Он может расширяться и сужаться при динамическом выделении и освобождении памяти. Для ссылочных классов определены четыре основные операции:

выбор:	X[ptr], где X — переменная типа $\mathbf{Ref}(\mathbf{T})$, а ptr — указатель типа $\mathbf{T}^*$;
присваивание:	$\langle \mathrm{X}, \mathrm{ptr}, \mathrm{value} \rangle$;
расширение:	$\mathrm{X} \cup \{\mathrm{address}_1, \mathrm{address}_2\},\ \mathrm{address}_1 \le \mathrm{address}_2$;
сужение:	$\mathrm{X} \setminus \{\mathrm{address}\}$.

Подробнее причины введения ссылочных классов рассмотрены в п. 4.2.

Алфавит языка спецификации состоит из следующих классов символов:

- *переменные*,
- *константы*,
- *кванторы* $\exists$ и $\forall$ и *логические связки* $\neg$, $\Rightarrow$, $\Leftrightarrow$, $\wedge$, $\vee$,
- *функциональные символы*,
- *предикатные символы*,
- *скобки* (,), [,], /%, %/, <, >, {, },
- *символы пунктуации*: точка, двоеточие, запятая,
- *символы операций*: символы всех операций C-light и $\cup$.

2.2.2. Выражения

Переменные могут быть любого типа, кроме функций и пустого типа. Константы могут быть любых типов, кроме ссылочных классов и пустого типа. Для любого типа $\mathbf{Ref}(T)$, где тип T — не пустой, может быть только одна переменная этого типа, которую обозначаем как $\mathbf{P}\#T$. Идентификаторы для всех остальных переменных строятся по обычным правилам построения идентификаторов языка C. Множество всех переменных обозначается идентификатором *Var*. Все переменные

базовых типов помимо значений соответствующих типов могут иметь неопределенные значения. Неопределенное значение обозначается как ω. Поскольку все составные типы в языке C являются типами второго порядка и выше, непосредственное сравнение переменных этих типов с конкретными значениями не имеет смысла.

Замечание. Выражения вида `a[i]`, `r.f`, `P#T[ptr]` также называем переменными, поскольку они могут находиться в левой части операции присваивания и быть аргументами в подстановке. По терминологии [5] это *subscripted variables*.

Поскольку выражения языка C-light-kernel существенно проще чем выражения языка C-light, то и статическая семантика (т. е. система типов) выглядит проще. Аксиомы для базовых типов приведены в работе [2]. Выражения языка спецификаций определяются по индукции следующим образом:

• переменная типа T является выражением типа T, если T не является массивом;

• если A — переменная типа T[n], то A — выражение типа T*;

• константа базового типа T является выражением типа T;

• если $s_1, ..., s_n$ — являются выражениями типов $T_1, ..., T_n$ соответственно, а op — константа типа $T_1 \times ... \times T_n \to T$, то $op(s_1, ..., s_n)$ является выражением типа T;

• если exp — выражение типа int, а A — переменная типа T[n], то A[exp] является выражением типа T;

• если f — переменная типа $\mathsf{struct}(T_1 v_1; ...; T_n v_n)$, то $f.v_i$ является выражением типа T_i;

• если P#**T** — переменная типа $\mathbf{Ref}(T)$, а ptr_1 — переменная типа T*, то P#T[ptr_1] является выражением типа T, а P#$T \cup \{ptr_1, ptr_2\}$ и P#$T \setminus \{\text{ptr}\}$ являются выражениями типа $\mathbf{Ref}(T)$;

• если v — переменная типа T_1, то $(T_2)v$ — выражение типа T_2 , если приведение от типа T_1 к типу T_2 безопасно.

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

2.2.3. Подстановка

Важным свойством языка спецификаций является возможность записи подстановок в выражениях. Подстановка — это функция, отобра-

жающая выражения в выражения и утверждения в утверждения. Как и в логике, подстановка терма вместо переменной — это просто замена всех свободных вхождений.

Очевидно, что с помощью подстановки можно моделировать присваивание значений программным переменным. Таким образом при обработке операции присваивания `x = e;`, где `x` — переменная базового типа, в некотором утверждении произойдет замена свободных вхождений переменной `x` на `e`. Однако в общем случае в выражении могут быть переменные составных типов. Если, например, есть присваивание `a[i] = e;`, то нельзя просто заменить *subscripted* переменную `a[i]` на терм `e`, поскольку изменяется весь массив `a`. Выход — заменить идентификатор `a` на конструкцию `upd⟨a, i, e⟩`, предложенную McCarthy [12]. По определению `upd⟨a, i, e⟩` — это массив, совпадающий с `a` всюду, кроме, может быть, `i`-й компоненты и `upd⟨a, i, e⟩[i] = e`, а для любого `j≠i upd⟨a, i, e⟩[j] = a[j]`. Аналогичную замену предлагается использовать для записей и ссылочных классов. Т. е., если есть присваивание `r.f = e;`, то идентификатор `r` заменяется на конструкцию `updr⟨r, f, e⟩`. Если есть присваивание `*ptr = e;`, где `ptr` — указатель типа T*, то изменяется ссылочный класс `P#T`, поэтому идентификатор `P#T` заменяется на конструкцию `upd⟨P#T, ptr, e⟩`. Префикс `upd` иногда опускается для удобства.

Подстановка выражения t вместо переменной u в выражении s обычно записывается в виде

$$s(u \leftarrow t)$$

и определяется индукцией по структуре выражения s (для краткости не рассматриваются стандартные правила подстановки в логических выражениях).

- если s — переменная базового типа или массив (запись, ссылочный класс), то

$$s(u \leftarrow t) \equiv \begin{cases} t, & \text{если } s \equiv u \\ s, & \text{иначе;} \end{cases}$$

- если s — константа базового типа, то
  $s(u \leftarrow t) \equiv s$;
- если $s \equiv op(s_1, ..., s_n)$ для некоторой константы-функции op, то
  $s(u \leftarrow t) \equiv op(s_1(u \leftarrow t), ..., s_n(u \leftarrow t))$;
- если $u \equiv \mathrm{A}[t_1])$, то

$s(u \leftarrow t) \equiv s(\ \leftarrow \mathrm{upd}\langle \mathrm{A}, t_1, t\rangle)$;

- если $u \equiv \mathrm{r.f}$, то
  $s(u \leftarrow t) \equiv s(\mathrm{r} \leftarrow \mathrm{updr}\langle \mathrm{r}, \mathrm{f}, t\rangle)$;
- если $u \equiv *\mathrm{ptr}$, где ptr — указатель типа T, то
  $s(u \leftarrow t) \equiv s(\mathrm{P\#T} \leftarrow \mathrm{updr}\langle \mathrm{P\#T}, \mathrm{ptr}, t\rangle)$;
- если $s \equiv (T)v$, то
  $s(u \leftarrow t) \equiv (T)v(u \leftarrow t)$.

Непосредственно из определения подстановки вытекает следующее утверждение.

Лемма 1. $s(u \leftarrow t) \equiv s$ если s не содержит u.

3. ОПЕРАЦИОННАЯ СЕМАНТИКА ЯЗЫКА C-LIGHT-KERNEL

Теоретической базой для верификации программ на языке программирования является формальное определение смысла конструкций языка. В нашем случае используются два подхода — операционный и аксиоматический. Формальным определением языка C-light-kernel является структурная операционная семантика, предложенная в работе [15]. В терминах этой семантики определяется понятие частичной корректности. Доказательство свойства частичной корректности для программ происходит на основе аксиоматической семантики. При создании семантик также использовались работы [5, 14].

Главной целью при создании операционной семантики языка C-light-kernel, в отличие от семантики базового языка C-light [2], являлось максимально возможное упрощение. Поэтому состояния абстрактной машины языка C-light-kernel содержат меньше компонент, а число отношений перехода уменьшено с трех до одного.

3.1. Состояния абстрактной машины и интерпретация выражений

Семантика — это отображение, которое сопоставляет каждому элементу из синтаксической области определения значение или интерпретацию, т. е. элемент семантической области. Сначала определим семантику выражений. Интерпретация выражения s обозначается как $\mathcal{I}\|s\|$. Это определение требует нескольких предварительных.

Во-первых, для каждого типа $\boldsymbol{T}$ фиксируем множество значений, называемое *носителем* типа $\boldsymbol{T}$ и обозначаемое D_T. Поскольку семантика не ориентирована на конкретную архитектуру или реализацию языка C, то границы носителей типов задаются символическими константами.

- $D_{\textbf{bool}} = \{\text{FALSE},\ \text{TRUE}\}$,
- $D_{\textbf{unsigned char}} = \{0 \ldots \text{MAX\_UNSIGNED\_CHAR}\}$,
- $D_{\textbf{signed char}} = \{\text{MIN\_SIGNED\_CHAR} \ldots \text{MAX\_SIGNED\_CHAR}\}$,
- $D_{\textbf{unsigned short}} = \{0 \ldots \text{MAX\_UNSIGNED\_SHORT}\}$,
- $D_{\textbf{signed short}} = \{\text{MIN\_SIGNED\_SHORT} \ldots \text{MAX\_SIGNED\_SHORT}\}$,
- $D_{\textbf{unsigned int}} = \{0 \ldots \text{MAX\_UNSIGNED\_INT}\}$,
- $D_{\textbf{signed int}} = \{\text{MIN\_SIGNED\_INT} \ldots \text{MAX\_SIGNED\_INT}\}$,
- $D_{\textbf{unsigned long}} = \{0 \ldots \text{MAX\_UNSIGNED\_LONG}\}$,
- $D_{\textbf{signed long}} = \{\text{MIN\_SIGNED\_LONG} \ldots \text{MAX\_SIGNED\_LONG}\}$,
- $D_{\textbf{wchar\_t}} = D_{\textbf{unsigned short}}$,
- $D_{\textbf{float}} = \{\text{MIN\_FLOAT} \ldots \text{MAX\_FLOAT}\}$,
- $D_{\textbf{double}} = \{\text{MIN\_DOUBLE} \ldots \text{MAX\_DOUBLE}\}$,
- $D_{\textbf{long double}} = \{\text{MIN\_LONG\_DOUBLE} \ldots \text{MAX\_LONG\_DOUBLE}\}$,
- $D_{\textbf{enum}} = D_{\textbf{signed int}}$ для любого перечисления,
- $D_{\textbf{void}} = \emptyset$,
- $D_{T*} = D_{\textbf{unsigned int}}$ для любого типа $\boldsymbol{T}$,
- $D_{T[\textbf{n}]} = D_T^n$ (декартова степень n) для любого непустого и нефункционального типа $\boldsymbol{T}$,
- $D_{\textbf{struct}(T_1\ v_1;\ \ldots;\ T_n\ v_n)} = D_{T_1} \times \ldots \times D_{T_n}$,
- $D_{\textbf{Ref}(T)} = D_{\textbf{unsigned int}} \to D_T$, т. е. множество всех функций из указателей в значения типа $\boldsymbol{T}$,
- $D_{T_1 \times \ldots \times T_n \to T} = D_{T_1} \times \ldots \times D_{T_n} \to D_T$, т. е. множество всех функций из декартова произведения множеств $D_{T_1}, \ldots, D_{T_n}$ во множество D_T.

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

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

Далее для любой константы типа $\boldsymbol{T}$ фиксируем значение в носителе D_T и говорим, что константа *обозначает* это значение. Считаем, что каждая константа базового типа (а также типов массив и структура) обозначает саму себя. В свою очередь константы типов функций обозначают соответствующие функции.

В отличие от констант, значения переменных не фиксированы и определяются через состояния абстрактной вычислительной машины.

Состояние — это в простейшем случае отображение, которое присваивает любой переменной типа $\boldsymbol{T}$ значение в области D_T. Для обозначения множества всех состояний используем слово *States*. В следующем пункте понятие состояния будет расширено так, что помимо отображения содержимого абстрактной памяти в нем будут дополнительные компоненты. Степень абстрактности вычислительной машины определяется, в первую очередь, целями, поставленными при верификации. Как будет видно из следующих пунктов, степень абстрактности нашей операционной семантики достаточно высока, например, мы не пытаемся моделировать какую-либо конкретную архитектуру и не изучаем раскладку данных в памяти.

Семантика $\mathcal{I}\|s\|$ выражения s типа $\boldsymbol{T}$ это отображение

$$\mathcal{I}\|s\| : States \to D_T,$$

определяемое индукцией по структуре s (рассмотрим ограниченное подмножество выражений):

- если s — переменная, то $\mathcal{I}\|s\|(\sigma) = \sigma(s)$;
- если s — константа базового типа, обозначающая значение d, то $\mathcal{I}\|s\|(\sigma) = d$;
- если $s \equiv op(s_1, ..., s_n)$ для некоторой константы op, обозначающей функцию f, то $\mathcal{I}\|s\|(\sigma) = f(\mathcal{I}\|s_1\|(\sigma), ..., \mathcal{I}\|s_n\|(\sigma))$;
- если $s \equiv (T)e$ для некоторого выражения e типа $'$, то $\mathcal{I}\|s\|(\sigma) = \gamma_{T',T}(\mathcal{I}\|e\|(\sigma))$, где $\gamma_{T',T}$ — отображение из типа T' в тип T.

Следует отметить, что если все выражения записывать в префиксной форме, то этих четырех случаев достаточно для всего языка. Например, бинарную операцию "+"можно считать константой типа $\mathbf{int} \times \mathsf{int} \to \mathsf{int}$, а операцию индексации в массиве "[]"можно считать константой типа $\mathbf{T}[\mathbf{n}] \times \mathsf{int} \to \mathbf{T}$.

Поскольку далее $\mathcal{I}$ всюду фиксировано, будем вместо $\mathcal{I}\|s\|(\sigma)$ писать просто $\sigma(s)$.

3.2. Значение логических утверждений

Определим *обновление* состояния σ, записываемое как $\sigma(u \leftarrow d)$, где u — переменная типа T, а d — элемент типа T. Это состояние, совпадающее с σ всюду кроме, может быть, переменной u, и $\sigma(u) = d$. Определение обновлений необходимо для моделирования значений присваиваний переменным.

Наконец, определим фундаментальное понятие *истинности* логического утверждения p в состоянии σ, записываемой в виде $\sigma \models p$. Истинность определяется индукцией по структуре утверждения p.

- $\sigma \models B$ iff $\sigma(B) = \text{TRUE}$, если B — элементарная логическая формула;
- $\sigma \models \neg p$ iff неверно, что $\sigma \models p$ (записывается в виде $\sigma \not\models p$);
- $\sigma \models p \vee q$ iff $\sigma \models p$ или $\sigma \models q$;
- $\sigma \models \exists x : p$ iff $\sigma(x \leftarrow d) \models p$, для некоторого элемента d из типа соответствующего x.

Для остальных логических связок и квантора $\forall$ истинность определяется из обычных логических соотношений ($p \wedge q \equiv \neg(\neg p \vee \neg q)$ и др).

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

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

Говорим, что утверждение p *истинно* или *выполняется*, если $\|p\| = States$. Следующая простая лемма описывает свойства стандартных булевских операций.

Лемма 2.

1. $\|\neg p\| = States - \|p\|$.
2. $\|p \vee q\| = \|p\| \cup \|q\|$.
3. $\|p \wedge q\| = \|p\| \cap \|q\|$.
4. $p \rightarrow q$ истинно iff $\|p\| \subseteq \|q\|$.
5. $p \leftrightarrow q$ истинно iff $\|p\| = \|q\|$.

В операционной семантике языка C-light-kernel особое значение имеет предикат $\mathcal{D}om(exp)$, где exp — произвольное неконстантное выражение языка C-light-kernel. Истинность этого предиката для конкретного выражения означает, что выражение находится в области его определенности. Естественно значения переменных определяются состоянием, поэтому истинность утверждения $\mathcal{D}om(exp)$ всегда зависит от состояния. Цель введения этого предиката состоит в том, чтобы сразу находить ошибки периода исполнения, например деление на ноль. В классических работах определенность выражений обычно не проверяется и работа абстрактной машины при ошибке не прерывается, поэтому вычисление выражений в заключительном состоянии может приводить к неопределенным значениям. В семантике языка C-light [2] выражения проверялись и для неопределенности использовался символ Ω. Однако вводить неопределенность в аксиоматическую семантику нежелательно,

поэтому в данной работе используется другой подход и ложное значение $\mathcal{D}om(exp)$ будет приводить к остановке программы (см. далее).

3.3. Лемма о подстановке

Пусть Z — некоторое множество переменных. Через $\sigma[Z]$ обозначаем сужение состояния σ на переменные из множества Z. Говорим, что множества состояний X и Y равны по модулю Z, и пишем $X = Y$ **mod** Z, если $\{\sigma[Var - Z] \mid \sigma \in X\} = \{\sigma[Var - Z] \mid \sigma \in Y\}$. Следующая лемма очевидно следует из определений.

Лемма 3. Для любого утверждения p, выражения s и состояний σ и τ

1. Если $\sigma[var(s)] = \tau[var(s)]$, то $\sigma(s) = \tau(s)$.
2. Если $\sigma[free(p)] = \tau[free(p)]$, то $\sigma \models p$ iff $\tau \models p$,

где $var(s)$ — множество переменных выражения s, а $free(p)$ — множество свободных переменных утверждения p.

Следующая лемма будет использоваться при доказательстве непротиворечивости правил аксиоматической семантики для присваивания. Она связывает понятия подстановки и обновления введенные ранее.

Лемма 4 (о подстановке). Для любого утверждения p, выражений s и t, простой или индексной переменной u того же типа, что и t, и состояния σ

1. $\sigma(s(u \leftarrow t)) = \sigma(u \leftarrow \sigma(t))(s)$.
2. $\sigma \models p(u \leftarrow t)$ iff $\sigma(u \leftarrow \sigma(t)) \models p$.

Доказательство.

1. Доказательство проводится индукцией по структуре s.

Пусть s — простая переменная или идентификатор массива, структуры, ссылочного класса.

Если $s \equiv u$, то $\sigma(s(u \leftarrow t)) = \sigma(t)$ по определению подстановки. По определению обновления $\sigma(t) = \sigma(s \leftarrow \sigma(t))(s)$. Так как $s \equiv u$, то $\sigma(s \leftarrow \sigma(t))(s) = \sigma(u \leftarrow \sigma(t))(s)$.

Если $s \neq u$, то $\sigma(s(u \leftarrow t)) = \sigma(s)$ по определению подстановки. По определению обновления $\sigma(s) = \sigma(u \leftarrow \sigma(t))(s)$.

Если $s \equiv op(s_1, ..., s_n)$ для некоторой константы op, обозначающей функцию f, то по определению подстановки и семантики $\sigma(s(u \leftarrow t)) = \sigma(op(s_1(u \leftarrow t), ..., s_n(u \leftarrow t))) = f(\sigma(s_1(u \leftarrow t)), ..., \sigma(s_n(u \leftarrow t)))$. По предположению индукции и определению семантики

$f(\sigma(s_1(u \leftarrow t)), ..., \sigma(s_n(u \leftarrow t))) =$
$f(\sigma(u \leftarrow s(t))(s_1), ..., \sigma(u \leftarrow s(t))(s_n)) =$
$\sigma(u \leftarrow \sigma(t))(s)$.

Если записывать выражения в префиксной форме, то последний случай охватывает позиционирование в массивах, структурах и ссылочных классах. Однако для лучшего понимания, рассмотрим случай массива подробно. Доказательство для структур и ссылочных классов совершенно аналогично, только для структур используется операция выбора элемента (точка) и конструкция $\mathrm{updr}\langle \mathrm{r}, \mathrm{f}, \mathrm{e}\rangle$, а для ссылочных классов — идентификатор P#T.

Пусть $s \equiv \mathrm{A}[s_1]$. Если u — простая переменная или $u \equiv \mathrm{B}[t_1]$, где $\mathrm{A} \neq \mathrm{B}$, то

$\sigma(s(u \leftarrow t))$
$=$ {опр. подстановки}
$\sigma(\mathrm{A}(u \leftarrow t)[s_1(u \leftarrow t)]) = \sigma(\mathrm{A}[s_1(u \leftarrow t)])$
$=$ {опр. семантики}
$\mathrm{A}[\sigma(s_1(u \leftarrow t))]$
$=$ {предположение индукции}
$A[\sigma(u \leftarrow \sigma(t))(s_1)]$
$=$ {по опр. обновления $\sigma(u \leftarrow \sigma(t))(\mathrm{A}) = \sigma(\mathrm{A})$}
$\sigma(u \leftarrow \sigma(t))(\mathrm{A})[\sigma(u \leftarrow \sigma(t))(s_1)]$
$=$ {опр. семантики, $s \equiv \mathrm{A}[s_1]$}
$\sigma(u \leftarrow \sigma(t))(s)$.

Если $u \equiv \mathrm{A}[t_1]$, то
$\sigma(s(u \leftarrow t))$
$=$ {опр. подстановки, $s \equiv \mathrm{A}[s_1], u \equiv \mathrm{A}[t_1]$}
$\sigma(\mathrm{upd}\langle \mathrm{A}, t_1, t\rangle[s_1(u \leftarrow t)])$
$=$ {опр. семантики, опр. upd}
$\begin{cases} \sigma(t), & \text{если } \sigma(s_1(u \leftarrow t)) = \sigma(t_1), \\ \sigma(\mathrm{A}[\sigma(s_1(u \leftarrow t))]), & \text{иначе} \end{cases}$
$=$ {опр. обновления, $u \equiv \mathrm{A}[t_1]$}
$\sigma(\mathrm{A} \leftarrow \mathrm{upd}\langle \mathrm{A}, \sigma(t_1), \sigma(t)\rangle)(\mathrm{A}[s_1(u \leftarrow t)]) = \sigma(u \leftarrow \sigma(t))(\mathrm{A}[s_1(u \leftarrow t)])$
$=$ {предположение индукции}
$\sigma(u \leftarrow \sigma(t))(\mathrm{A}[\sigma(u \leftarrow \sigma(t))(s_1)])$
$=$ {опр. семантики, $s \equiv \mathrm{A}[s_1]$}
$\sigma(u \leftarrow \sigma(t))(s)$.

Для приведения типов доказательство очевидно.

2. Индукция по структуре утверждения p. Очевидно, что базис (элементарные формулы) фактически доказан в пункте 1. Шаг индукции также очевиден для разных логических связок кроме случая $p \equiv \exists \mathrm{x} : r$. Пусть y — простая переменная, которая не появляется в r, t или u и того же типа, что и x. Тогда

$\sigma \models (\exists \mathrm{x} : r)(u \leftarrow t)$
iff {опр. подстановки}
$\sigma \models \exists y : r(x \leftarrow y)(u \leftarrow t)$
iff {опр. истинности}
$\sigma' \models r(x \leftarrow y)(u \leftarrow t)$ для некоторого d из типа соответствующего y и $\sigma' = \sigma(y \leftarrow d)$
iff {предположение индукции}
$\sigma'(u \leftarrow \sigma'(t)) \models r(x \leftarrow y)$ для некоторого d и σ' как и выше
iff $\{y \neq x \Rightarrow \sigma'(u \leftarrow \sigma'(t))(y) = d$ и предположение индукции$\}$
$\sigma'(u \leftarrow \sigma'(t))(x \leftarrow d) \models r$, где d и σ' как и выше
iff {лемма 3, выбор y}
$\sigma(u \leftarrow \sigma(t))(x \leftarrow d) \models r$, для некоторого d как и выше
iff {опр. истинности}
$\sigma(u \leftarrow \sigma(t)) \models \exists \mathrm{x} : r$.

■

3.4. Конфигурации абстрактной машины

Операционная семантика языка программирования представляется в виде множеств пар конфигураций абстрактной вычислительной машины. Конфигурация — это пара $\langle P, \sigma \rangle$, где P — это программа, а σ — состояние. Частными случаями программ могут быть значения типов, объявленных в программе. Пустая программа обозначается символом E, и записи $E; A$ и $A; E$ являются эквивалентными обозначениями для программы A. Пустая программа необходима для определения завершения исполнения. Значения нужно рассматривать совместно с программами для того, чтобы упростить семантику вызовов функций, возвращающих значения (особенно рекурсивных).

В простейшем случае состояние — это отображение из множества идентификаторов переменных во множество их потенциальных значений. Такого представления достаточно для простых модельных языков [5], но для языка C-light-kernel, как и для исходного языка C-light, необходимо добавить новые компоненты.

Определение. Состояние абстрактной вычислительной машины языка C-light-kernel — это пятерка {M, Γ, Σ, Φ, TD}, где

1. M — отображение всех переменных программы во множество их потенциальных значений. Фактически это отображение памяти абстрактной машины.
2. Γ — информация о типах переменных, т. е. отображение из множества имен переменных программы в типы.
3. Σ — информация о структурах. Это отображение, сопоставляющее тегу структуры последовательность пар имен полей и типов полей структуры.
4. Φ — информация о функциях, т. е. отображение, сопоставляющее идентификатору процедуры кортеж типов ее аргументов и тип возвращаемого значения.
5. TD — информация о синонимах типов, т. е. отображение, сопоставляющее идентификатору имя уже существующего типа, связанное с ним typedef декларацией.

При определении конфигураций абстрактной вычислительной машины языка C-light-kernel основной целью было максимально возможное упрощение состояний. Например, по сравнению с семантикой языка C-light, отсутствуют очередь отложенных побочных эффектов и флаг вложенности [2]. Также нет явной работы со стеком, поскольку, во-первых, нет локальных переопределений, а во-вторых, на вход подается программа корректная с точки зрения стандартного компилятора языка C. Поэтому при обработке оператора goto можно безопасно (в семантике) раскрывать блоки.

В случаях, когда необходимо явно указать, что компонента X соответствует некоторому состоянию σ, будет использоваться запись X_σ.

Ранее уже определялось понятие обновления состояния в случае, когда состояние представляло собой одно отображение. Далее запись $\sigma(\mathrm{X}(a \leftarrow e))$ означает состояние σ', такое, что компонента X состояния σ' совпадает с компонентой X состояния σ всюду, кроме, может быть, a, и $\mathrm{X}_{\sigma'}(a) = e$, а все остальные компоненты совпадают полностью. Изменение нескольких компонент состояния, записывается через запятую: $\sigma(X(a \leftarrow e), Y(b \leftarrow i))$.

Итак, опишем все аксиомы и правила операционной семантики языка C-light-kernel. Аксиомы будут изображаться в виде пар конфигураций, связанных одним из отношений перехода. Тогда запись $\langle A, \sigma\rangle \rightarrow \langle B, \tau\rangle$ означает, что один шаг исполнения фрагмента A исходной про-

граммы, начинающийся в состоянии σ, приводит в состояние τ и B — тот фрагмент исходной программы, который остается для исполнения. Заметим, что если бы в исходной программе отсутствовали переходы "goto назад", то фрагмент B был бы частью A.

3.5. Декларации

В соответствии с формальным синтаксисом языка C-light-kernel [2] в программе можно декларировать спецификации, типы, классы (т. е. структуры и объединения), функции и все остальные объекты (переменные, члены структур и др.).

Аннотации. Аннотации используются в аксиоматической семантике и никак не влияют на ход работы программы, т. е. операционная семантика их просто игнорирует. Поэтому аксиома для аннотации выглядит как

$$\langle /\% \ \_condbody \ \%/, \sigma\rangle \to \langle E, \sigma\rangle. \tag{1}$$

Типы. Фактически происходит переименовывание:

$$\begin{aligned}&\langle \mathsf{typedef}\ \text{type\_spec IdLexem};, \sigma\rangle \to \\ &\quad \langle E, \sigma(\text{TD}(\text{IdLexem} \leftarrow \text{type\_spec}))\rangle.\end{aligned} \tag{2}$$

Структуры. При объявлении структуры тегу сопоставляется последовательность пар имен полей и типов полей структуры.

$$\langle \mathsf{struct}\ \text{TAG};, \sigma\rangle \to \langle E, \sigma(\Sigma(\text{TAG} \leftarrow \emptyset))\rangle. \tag{3}$$

$$\begin{aligned}&\langle \mathsf{struct}\ \text{TAG}\{T_1\ v_1; ...; T_n\ v_n\};, \sigma\rangle \to \\ &\quad \langle E, \sigma(\Sigma(\text{TAG} \leftarrow \{(v_1, T_1), ..., (v_n, T_n)\}))\rangle.\end{aligned} \tag{4}$$

Функции. Декларация функции может быть либо прототипом, либо определением. Так как работа программы на языке C и, соответственно, на языке C-light-kernel начинается с функции `main`, то тела всех других функций обрабатываются только при их вызовах. В текущей версии системы не рассматриваются аргументы командной строки, поэтому функции `main` не имеет аргументов argc, argv, env.

$$\begin{aligned}&\langle \mathsf{ret\_type}\ \text{fname}(T_1\ v_1, ..., T_n\ v_n);, \sigma\rangle \to \\ &\quad \langle E, \sigma(\Phi(\text{fname} \leftarrow)(\mathsf{ret\_type}, T_1, ..., T_n))\rangle.\end{aligned} \tag{5}$$

$$\langle \mathsf{ret\_type}\ \mathrm{fname}(T_1\ v_1, ..., T_n\ v_n)B, \sigma\rangle \rightarrow \langle E, \sigma(\Phi(\mathrm{fname} \leftarrow)(\mathsf{ret\_type}, T_1, ..., T_n))\rangle. \quad (6)$$

$$\langle \mathsf{ret\_type}\ \mathtt{main}()B, \sigma\rangle \rightarrow \langle B, \sigma(\Phi(\mathtt{main} \leftarrow (\mathsf{ret\_type})))\rangle. \quad (7)$$

Переменные. Чтобы не задавать правила объявления для всех типов и всех возможных модификаторов типов, используется только одна аксиома:

$$\langle \mathsf{data\_decln}, \sigma\rangle \rightarrow \langle E, \mathcal{D}ec(\sigma, \mathsf{data\_decln})\rangle, \quad (8)$$

а рекурсивная функция $\mathcal{D}ec$ определяется в соответствии с синтаксисом языка C-light-kernel (для краткости объявления переменных фундаментальных типов рассматриваются для типа int, что не ограничивает общности):

$\mathcal{D}ec$

$(\sigma, \mathsf{static\ int}\ v) = \sigma(\mathrm{M}(v \leftarrow 0), \Gamma(v \leftarrow \mathsf{int}));$

$(\sigma, \mathsf{auto\ int}\ v) = \sigma(\Gamma(v \leftarrow \mathsf{int}));$

$(\sigma, \mathsf{int}\ v = exp) = \sigma(\mathrm{M}(v \leftarrow \mathrm{M}_\sigma(exp)), \Gamma(v \leftarrow \mathsf{int}))$,
если $\sigma \models \mathcal{D}om(exp)$, иначе **fail**;

$(\sigma, \mathsf{static}\ \mathbf{T}*\ x) = \sigma(\mathrm{M}(x \leftarrow \mathrm{NULL}), \Gamma(x \leftarrow \mathbf{T}*));$

$(\sigma, \mathsf{auto}\ \mathbf{T}*\ x) = \sigma(\Gamma(x \leftarrow \mathbf{T}*));$

$(\sigma, \mathbf{T}*\ x = exp) = \sigma(\mathrm{M}(x \leftarrow \mathrm{M}_\sigma(exp)), \Gamma(x \leftarrow \mathbf{T}*))$,
если $\sigma \models \mathcal{D}om(exp)$, иначе **fail**;

$(\sigma, \mathsf{static}\ \mathbf{T}\ \mathrm{A[n]}) = \sigma(\mathrm{M}(0 \le i \le \mathrm{n}-1(\mathrm{P}\#\mathbf{T} \leftarrow \mathrm{M}_\sigma(\langle \mathrm{P}\#\mathbf{T}, \mathrm{A}+i, 0\rangle))),$
$\Gamma(\mathrm{A} \leftarrow \mathbf{T}*));$

$(\sigma, \mathsf{auto}\ \mathbf{T}\ \mathrm{A[n]}) = \sigma(\Gamma(\mathrm{A} \leftarrow \mathbf{T}*));$

$(s, \mathbf{T}\ \mathrm{A[n]} = \{l_0, ..., l_k\}) = \sigma(\mathrm{M}(0 \le i \le k(\mathrm{P}\#\mathbf{T} \leftarrow \mathrm{M}_\sigma(\langle \mathrm{P}\#\mathbf{T}, \mathrm{A}+i, l_i\rangle)))$
$(k < j \le \mathrm{n}-1(\mathrm{P}\#\mathbf{T} \leftarrow \mathrm{M}_\sigma(\langle \mathrm{P}\#\mathbf{T}, \mathrm{A}+j, 0\rangle))),$
$\Gamma(\mathrm{A} \leftarrow \mathbf{T}*));$

$(\sigma, \mathsf{static\ struct\ TAG}[\{...; \mathbf{T}_i\ v_i; ...\}]\ v) = \sigma(\mathrm{M}(v \leftarrow \mathrm{M}_\sigma(\mathrm{updr}\langle v, v_i, 0\rangle)),$

$$\Gamma(v \leftarrow \mathrm{TAG})),$$
$$\text{если } \Sigma_\sigma(\mathrm{TAG}) \neq \emptyset;$$

$$(\sigma, \textsf{auto struct } \mathrm{TAG}[\{...; \mathbf{T}_i\ v_i; ...\}]\ v) = \quad \sigma(\Gamma(v \leftarrow \mathrm{TAG})),$$
$$\text{если } \Sigma_\sigma(\mathrm{TAG}) \neq \emptyset;$$

$$(\sigma, \textsf{struct } \mathrm{TAG}[\{...; \mathbf{T}_i\ v_i; ...\}]\ v = \{..., l_i, ...\}) =$$
$$\sigma(\mathrm{M}(v \leftarrow \mathrm{M}_\sigma(\mathrm{updr}\langle v, v_i, l_i\rangle))),$$
$$\Gamma(v \leftarrow \mathrm{TAG})),$$
$$\text{если } \Sigma_\sigma(\mathrm{TAG}) \neq \emptyset;$$

$$(\sigma, \mathbf{T}(*a)\ \textsf{abs\_drator}) = \mathcal{D}ec(\sigma(\mathrm{TD}(\_\_a\_\mathrm{Type} \leftarrow \mathbf{T}\ \textsf{abs\_drator})),$$
$$\_\_\mathrm{a}\_\mathrm{Type}*\ a).$$

Ветви функции $\mathcal{D}ec$ естественным образом отражают правило начальной инициализации в языке C, принятое и в C-light-kernel: статические переменные инициализируются нулем, значение автоматической переменной не определено, пока ей не будет явно присвоено значение. При инициализации некоторым выражением проверяется истинность предиката $\mathcal{D}om$ в данном состоянии, в случае ложного значения происходит переход в специальное состояние **fail**. Напомним, что предикат $\mathcal{D}om$, введенный в п. 3.2, проверяет соответствие выражения его области определенности. Примером случая ложного значения предиката может служить ситуация, когда при вычислении выражения отображение M не определено на некоторой переменной, что соответствует попытке использования неинициализированной локальной переменной.

Следует отдельно пояснить последние четыре ветви функции $\mathcal{D}ec$. При объявлении переменной типа struct предполагается, что соответствующий тип TAG уже объявлен. В случае одновременного объявления типа TAG и переменной этого типа обработка для простоты разбивается на два шага: вначале срабатывает правило (4), а затем соответствующая ветвь функции $\mathcal{D}ec$. Последняя ветвь относится, например, к объявлению указателя на массив. Опять же на основе абстрактного декларатора abs_drator (в данном случае число в квадратных скобках) строится новый тип $\_\_a\_\mathrm{Type}$ — массив соответствующего размера. Далее просто объявляется переменная типа "указатель на $\_\_a\_\mathrm{Type}$". Объявления указателей на функции не рассматриваются, так как они запрещены.

3.6. Выражения

Как уже говорилось, одной из основных трудностей при создании семантики для языка C является наличие побочных эффектов в выражениях. И наибольшую проблему здесь представляют побочные эффекты в выражениях, которые либо являются аргументами функций, либо стоят, например, в условиях циклов или условного оператора. Однако мы избавляемся от такого использования выражений при переводе в C-light-kernel. Все выражения в условиях не содержат побочных эффектов, а аргументами функций являются либо константы, либо переменные.

Аксиомы для оператора-выражения и вызова функции приводятся в следующем пункте, а здесь рассмотрена функция $\mathcal{U}pd$, определяющая действие оператора присваивания. Сама эта функция вызывается в аксиоме для оператора присваивания. В следующем определении полагаем, что $rval$ — выражение без вызовов функций.

$\mathcal{U}pd$

$$(\mathrm{A[i]} = rval, \sigma) = \langle rval, \sigma(\mathrm{M}(\mathrm{P}\#\mathbf{T} \leftarrow \mathrm{M}_\sigma(\langle \mathrm{P}\#\mathbf{T}, \mathrm{A} + \mathrm{i}, rval\rangle)))\rangle, \text{ если } \Gamma_\sigma(\mathrm{A}) = \mathbf{T}* \text{ и } \sigma \models \mathcal{D}om(rval);$$

$$(\mathrm{A[i]} = rval, \sigma) = \langle E, \mathbf{fail}\rangle, \text{ если } \sigma \not\models \mathcal{D}om(rval);$$

$$(\mathrm{r.f} = rval, \sigma) = \langle rval, \sigma(\mathrm{M}(\mathrm{r} \leftarrow \mathrm{M}_\sigma(\mathrm{updr}\langle \mathrm{r}, \mathrm{f}, rval\rangle)))\rangle, \text{ если } \sigma \models \mathcal{D}om(rval);$$

$$(\mathrm{r.f} = rval, \sigma) = \langle E, \mathbf{fail}\rangle, \text{ если } \sigma \not\models \mathcal{D}om(rval);$$

$$(*\mathrm{x} = rval, \sigma) = \langle rval, \sigma(\mathrm{M}(\mathrm{P}\#\mathbf{T} \leftarrow \mathrm{M}_\sigma(\langle \mathrm{P}\#\mathbf{T}, \mathrm{x}, rval\rangle)))\rangle, \text{ если } \Gamma_\sigma(x) = \mathbf{T}* \text{ и } \sigma \models \mathcal{D}om(rval);$$

$$(*\mathrm{x} = rval, \sigma) = \langle E, \mathbf{fail}\rangle, \text{ если } \sigma \not\models \mathcal{D}om(rval);$$

$$(\mathrm{ptr} = \mathsf{new}\ \mathbf{T}, \sigma) = \langle val, \sigma'\rangle, \text{ где } \sigma' = \sigma(\mathrm{M}(\mathrm{ptr} \leftarrow val)\ (\mathrm{P}\#\mathbf{T} \leftarrow \mathrm{M}_\sigma(\mathrm{P}\#\mathbf{T} \cup \{val, val\}))\ \text{и } \sigma \not\models \mathrm{PnTo}(\mathrm{P}\#\mathbf{T}, val);$$

$$(\mathrm{ptr} = \mathsf{new}\ \mathbf{T}[expn], \sigma) = \langle val, \sigma'\rangle, \text{ где } \sigma' = \sigma(\mathrm{M}(\mathrm{ptr} \leftarrow val)\ (\mathrm{P}\#\mathbf{T} \leftarrow \mathrm{M}_\sigma(\mathrm{P}\#\mathbf{T} \cup \{val, val + expn - 1\}))\ \text{и } \sigma \not\models \mathrm{PnTo}(\mathrm{P}\#\mathbf{T}, val);$$

$$
\begin{array}{lcl}
(lval = rval, \sigma) & = & \langle rval, \sigma(\mathrm{M}(lval \leftarrow \mathrm{M}_\sigma(rval)))\rangle, \\
 & & \text{если } \sigma \models \mathcal{D}om(rval), \text{ иначе } \langle E, \mathbf{fail}\rangle; \\
 & & \\
(lval = f(e_1, ...e_n), \sigma) & = & \mathcal{U}pd(lval = result, \sigma'), \\
 & & \text{где } \langle f(e_1, ...e_n);, \sigma\rangle \rightarrow^* \langle result, \sigma'\rangle, \\
 & & \text{если } \sigma \models \mathcal{D}om(f(e_1, ...e_n)); \\
 & & \\
(lval = f(e_1, ...e_n), \sigma) & = & \langle E, \mathbf{fail}\rangle, \text{ если } \sigma \not\models \mathcal{D}om(f(e_1, ...e_n));
\end{array}
$$

Ветви функции для присваивания элементу массива и разыменованному указателю отражают изменение соответствующих ссылочных классов, введенных в п. 2.2.1. При выделении динамической памяти указателю присваивается значение отличное от значений всех указателей, для которых ранее вызывалась операция new. Уникальность нового адреса моделируется ложностью предиката PnTo, который подробно рассмотрен в п. 4.2.

Особый интерес представляют две последние ветви функции $\mathcal{U}pd$, которые определяют обработку вызова функции, возвращающей значение, как обработку вызова процедуры с запоминанием значения в первой компоненте конфигурации, соответствующей окончанию обработки тела функции.

3.7. Операторы

Пустой оператор. Аксиома для пустого оператора очевидна:

$$\langle ;, \sigma\rangle \rightarrow \langle E, \sigma\rangle. \tag{9}$$

Оператор-объявление. Семантика деклараций рассмотрена ранее, и аксиома для оператора-объявления просто вызывает функцию $\mathcal{D}ec$:

$$\langle \mathsf{data\_decln};, \sigma\rangle \rightarrow \langle E, \mathcal{D}ec(\sigma, \mathsf{data\_decln})\rangle. \tag{10}$$

Оператор-выражение. В языке C-light-kernel оператором-выражением является либо оператор присваивания, либо вызов процедуры, либо операция освобождения памяти. Несмотря на то, что выражения при переписывании упрощаются, без их дополнительного анализа не обойтись, так как в них могут содержаться вызовы функций, возвращающих значения. Основная проблема в том, что если при вычислении,

например, выражения `x = sin(y);`, сделать подстановку терма `sin(y)` вместо переменной `x`, то доказать полученное условие корректности в процессе верификации не удастся, так как `sin` — программная функция, а не функция языка спецификаций. В языке спецификаций функциями считаются только допустимые в C-light-kernel стандартные операции. Поэтому при вызове любой другой функции (библиотечной либо пользовательской) подставляться должно не символическое значение `f(y)`, а реальное значение, полученное при исполнении тела функции.

Так как вызов функции подразумевает переход к ее телу, то нахождение возвращаемого значения отделяется от его дальнейшего присваивания, поэтому возникает вопрос — где это значение должно запоминаться. Самый простой способ — введение новых глобальных переменных, которым в телах функций будут присваиваться возвращаемые значения. Однако, во-первых, необходимо проверять уникальность этих переменных[1], а во-вторых, эти глобальные переменные повлекут появление в правилах аксиоматической семантики кванторов всеобщности. Поэтому, как и в языке C-light [2], конфигурации становятся неоднородными и первым элементом может быть не только программа, но и значение одного из объявленных в программе типов.

Таким образом, в определении функции $\mathcal{U}pd$ в предпоследней ветви сперва обрабатывалось тело функции, а затем полученное значение присваивалось левой части выражения присваивания. При этом переход к телу происходит по общей для функций и процедур аксиоме (12). Отметим, что транслятор сокращает число вызовов функций в одном выражении до одного. Сама аксиома для оператора вычисления выражения, в которой происходит вызов функции $\mathcal{U}pd$ выглядит так:

$$\langle \mathsf{expn};, \sigma \rangle \to \mathcal{U}pd(\mathsf{expn}, \sigma). \tag{11}$$

Вызов процедуры обрабатывается следующим образом:

$$\langle \mathrm{f}(a_1, ..., a_n);, \sigma \rangle \to \langle B(f_1 \leftarrow a_1, ..., f_n \leftarrow a_n), \sigma \rangle, \tag{12}$$

где B — тело процедуры f, f_i – формальные параметры, a_i — фактические аргументы.

Для освобождения динамической памяти используются две аксиомы:

$$\langle \mathsf{delete}\ \mathrm{ptr};, \sigma \rangle \to \langle E, \sigma(\mathrm{M}(\mathrm{P}\#\mathbf{T} \leftarrow \mathrm{M}_\sigma(\mathrm{P}\#\mathbf{T} \setminus \{\mathrm{ptr}\}))) \rangle, \quad \text{если } \mathrm{PnTo}(\mathrm{P}\#\mathbf{T}, \mathrm{ptr}). \tag{13}$$

[1]Это особенно актуально для рекурсивных функций.

$$\langle \mathsf{delete}\ []\ \mathrm{ptr};, \sigma\rangle \to \langle E, \sigma(\mathrm{M}(\mathrm{P}\#\mathbf{T} \leftarrow \mathrm{M}_\sigma(\mathrm{P}\#\mathbf{T} \setminus \{\mathrm{ptr}\})))\rangle,$$
$$\text{если } \mathrm{PnTo}(\mathrm{P}\#\mathbf{T}, \mathrm{ptr}). \qquad (14)$$

Оператор перехода. Большие трудности в структурной семантике вызывает оператор перехода goto. Для определения фрагмента исходной программы, на начало которого передается управление, используется функция $\mathcal{L}abel$.

$$\langle \mathsf{goto}\ \mathrm{L}, \sigma\rangle \to \langle \mathcal{L}abel(\mathrm{L}), \sigma\rangle. \qquad (15)$$

Поскольку в пределах тела одной функции управление может быть передано куда угодно, то определение функции $\mathcal{L}abel$ может быть очень сложным. Поэтому в определении языка было сформулировано важное ограничение — запрет на передачу управления внутрь любого блока. Более того, без такого ограничения состояние после перехода может измениться, т. е. аксиома не соответствует реальному поведению программы. Применяется функция $\mathcal{L}abel$ к текущей программной функции:

$$\mathcal{L}abel(\mathrm{L})(A\ L : B\ C)\ =\ B\ C,$$

где B — оператор (возможно пустой), а A и C — фрагменты программы (также возможно пустые). Ограничение на переход внутрь блоков означает, что если в A есть открывающая фигурная скобка, а в C — парная ей закрывающая, то инструкция `goto L` обязательно находится в этих пределах. При этом возможна ситуация, когда фрагмент $B\ C$ содержит закрывающие скобки без парных им открывающих. Как отмечалось при определении конфигураций, раскрытие блоков безопасно, поэтому эти скобки будут просто игнорироваться.

Условный оператор. Поскольку в зависимости от значения условного выражения в этом операторе выполняется одна из двух альтернативных ветвей, то его семантика задается двумя аксиомами:

$$\langle \mathsf{if}\ (\alpha)\ A\ \mathsf{else}\ B, \sigma\rangle \to \langle A, \sigma\rangle \text{ при } \sigma \models \alpha. \qquad (16)$$

$$\langle \mathsf{if}\ (\alpha)\ A\ \mathsf{else}\ B, \sigma\rangle \to \langle B, \sigma\rangle \text{ при } \sigma \models \neg\alpha. \qquad (17)$$

Оператор выбора. Этот оператор тоже определяется схемой из двух аксиом, причем первая аксиома задается в параметрическом виде, где роль параметра играет число ветвей:

$$\langle \mathsf{switch}\ (z)\{\ldots\ \mathsf{case}\ k_i : A_i\ \ldots\ \mathsf{default} : B\}, \sigma\rangle \to \langle A_i, \sigma\rangle$$
$$\text{при } \sigma \models (z = k_i). \qquad (18)$$

$$\langle \mathsf{switch}\ (z)\{\ldots\ \mathsf{case}\ k_i : A_i\ \ldots\ \mathsf{default} : B\}, \sigma\rangle \to \langle B, \sigma\rangle \quad \text{при } \sigma \models (z \notin \{\ldots, k_i, \ldots\}). \tag{19}$$

Оператор цикла. Также используются две аксиомы:

$$\langle \mathsf{while}\ (\alpha)\ A, \sigma\rangle \to \langle A; \mathsf{while}\ (\alpha)\ A, \sigma\rangle \text{ при } \sigma \models \alpha. \tag{20}$$

$$\langle \mathsf{while}\ (\alpha)\ A, \sigma\rangle \to \langle E, \sigma\rangle \text{ при } \sigma \models \neg\alpha. \tag{21}$$

Композиция операторов.

$$\frac{\langle A_1, \sigma\rangle \to \langle A_2, \tau\rangle}{\langle A_1\ A, \sigma\rangle \to \langle A_2\ A, \tau\rangle}. \tag{22}$$

Блок. Аксиома просто раскрывает фигурные скобки. Это действие может показаться лишним, но если бы мы просто двигались от скобки к парной ей скобке, то для того чтобы обрабатывать вложенность, семантику пришлось бы сильно усложнять. Поэтому сначала целиком считываем блок, а потом переходим к его содержимому.

$$\langle \{A\}, \sigma\rangle \to \langle A; , \sigma\rangle. \tag{23}$$

Помеченный оператор. Метка перед оператором никак не влияет на исполнение этого оператора:

$$\langle L :\ A, \sigma\rangle \to \langle A, \sigma\rangle. \tag{24}$$

Аварийное завершение. В аксиомах для деклараций и выражений упоминалось специальное состояние **fail**, которое моделирует ошибки времени исполнения, например деление на ноль. В классических работах такие ситуации обычно не рассматриваются, т. е. программа либо зацикливается, либо завершает работу в состоянии, о котором ничего нельзя сказать заранее. В частности это означает, что в таких работах неопределенность в значениях выражений протягивается до конца вычислений. Однако работать в логике с неопределенностью достаточно сложно. Поэтому предлагается использовать специальное состояние — **fail**, переход в которое означает, что в программе произошла фатальная ошибка. Для того, чтобы работа абстракной машины при этом действительно завершилась вводится следующая аксиома:

$$\langle A, \mathbf{fail}\rangle \to \langle E, \mathbf{fail}\rangle, \tag{25}$$

где A — любая непустая программа.

3.8. Свойства семантики. Частичная корректность

Таким образом цепочки конфигураций абстрактной машины определяют исполнения программ. Поскольку нас не интересует вопрос завершимости, то специальное «несобственное» состояние, обозначающее зацикливание, не вводится. Заметим, что хотя эта семантика имеет дело с состояниями памяти, это все же семантика высокого уровня (в отличие от языка C-light), так как операторы-объявления, операторы-выражения и оценки логических выражений выполняются за один шаг.

Определяя транзитивное, рефлексивное замыкание ($\rightarrow^*$) для отношения перехода, можем определить семантику программы S как отображение из $Spaces$ в 2^{Spaces}:

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

где $Final$ — пустая программа или значение некоторого программного типа. Т. е. семантика рассматривает все возможные исполнения программы, начинающиеся из состояния σ. Отображение $\mathcal{M}$ называется ***семантикой частичной корректности*** для языка C-light-kernel. Также по определению

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

где P — формула.

Непосредственно из определения аксиом и правил представленной семантики следуют два важных свойства:

1. Для любой конфигурации существует не более одного наследника в отношении перехода.
2. Для любой конфигурации $\langle S, \sigma\rangle$, где $S \neq E$ существует конфигурация $\langle S_1, \tau\rangle$ такая, что $\langle S, \sigma\rangle \rightarrow \langle S_1, \tau\rangle$.

Как следствие первого свойства имеем $|\mathcal{M}[\![S]\!](\sigma)| \leq 1$.

Лемма 5. Для семантики частичной корректности верны следующие свойства:

(a) Монотонность, т.е. если $X \subseteq Y \subseteq Spaces$, то $\mathcal{M}[\![A]\!](X) \subseteq \mathcal{M}[\![A]\!](Y)$;

(b) $\mathcal{M}[\![A_1\ A_2]\!](X) = \mathcal{M}[\![A_2]\!](\mathcal{M}[\![A_1]\!](X))$
$\mathcal{M}[\![(A_1\ A_2)\ A_3]](X) = \mathcal{M}[\![A_1\ (A_2\ A_3)]\!](X)$;

(c) $\mathcal{M}[\![\mathsf{if}\ (\alpha)\ A_1\ \mathsf{else}\ A_2]\!](X) = \mathcal{M}[\![A_1]\!](X \cap \|\alpha\|) \cup \mathcal{M}[\![A_2]\!](X \cap \|\neg\alpha\|)$;

(d) $\mathcal{M}[\![\mathsf{switch}\ (\alpha)\{...;\ \mathsf{case}\ k_i : A_i ...;\ \mathsf{default} : B\}]\!](X) =$
$\bigcup\limits_i \mathcal{M}[\![A_i]\!](X \cap \|\alpha = k_i\|) \cup \mathcal{M}[\![B]\!](X \cap \|\alpha \notin \{...\, k_i \, ...\}\|)$;

(e) $\mathcal{M}[\![\mathsf{while}\ (\alpha)\ A]\!] = \bigcup\limits_{k=0}^{\infty} \mathcal{M}[\![(\mathsf{while}\ (\alpha)\ A)^k]\!]$,

где $(\mathsf{while}\ (\alpha)\ A)^0 = \mathsf{while}\ (1);$ // т. е. бесконечный цикл
$(\mathsf{while}\ (\alpha)\ A)^{k+1} = \mathsf{if}\ (\alpha)\ \{A; (\mathsf{while}\ (\alpha)\ A)^k\}$.

Как уже упоминалось, операционная семантика, даже с высокой степенью абстрактности, неудобна для реальной верификации программ. Хоаром был предложен другой подход — работать не с отдельными состояниями, а с множествами состояний. Поскольку множества состояний можно определять формулами, истинными в этих множествах, то вход и выход программы характеризуются специальными формулами языка спецификаций — предусловием и постусловием. При этом для того чтобы обрабатывать циклы вне зависимости от числа итераций, для них используют свои формулы — инварианты циклов. Основной объект в этой семантике — это тройка Хоара $\langle\, P \,\rangle\, A \,\langle\, Q \,\rangle$, где A — некоторая программа, а P и Q — ее предусловие и постусловие соответственно. Фактически эта тройка является формулой, и она считается истинной в смысле частичной корректности, если из того, что утверждение P истинно перед исполнением программы и программа A завершается, следует, что утверждение Q истинно.

Формально понятие истинности тройки Хоара дается на основе операционной семантики частичной корректности:

Определение. Говорим, что тройка Хоара $\langle\, P \,\rangle\, A \,\langle\, Q \,\rangle$ истинна в смысле частичной корректности и изображаем $\models \langle\, P \,\rangle\, A \,\langle\, Q \,\rangle$, если

$$\mathcal{M}[\![A]\!](\|P\|) \subseteq \|Q\|.$$

4. АКСИОМАТИЧЕСКАЯ СЕМАНТИКА ЯЗЫКА C-LIGHT-KERNEL

Аксиоматическая семантика для любого языка программирования определяется как система вывода, в которой роль формул играют тройки Хоара. При этом программные конструкции являются преобразователями предикатов. При выводе тройки Хоара в этой системе происходит элиминация операторов, и истинность тройки сводится к истинности набора лемм — условий корректности, которые не содержат программных конструкций. Эти леммы интерпретируются в соответству-

ющих проблемных областях. Как будет показано в следующем разд., истинность этих лемм определяет истинность тройки Хоара для исходной программы. Не все конструкции императивных языков допускают удобные правила в стиле Хоара, некоторые вообще не аксиоматизируются. Это является одной из причин трансляции в язык C-light-kernel.

4.1. Система вывода HSC

Пусть символы P, Q, S,..., W обозначают аннотации, символы A, B (с индексами) — операторы; *DecStat* и *ExpStat* обозначают оператор-объявление и оператор-выражение соответственно. Вначале приведена вся система Хоара HSC для языка C-light-kernel, а далее подробно комментируются все аксиомы и правила. Аксиомы для работы с указателями рассматриваются отдельно в следующем пункте.

Пустой оператор.

$$\langle\, Q \,\rangle\,;\, \langle\, Q \,\rangle. \tag{A1}$$

Усиление посылки/ослабление заключения.

$$\frac{P \Rightarrow S \qquad \langle\, S \,\rangle\, A \,\langle\, T \,\rangle \qquad T \Rightarrow Q}{\langle\, P \,\rangle\, A \,\langle\, Q \,\rangle}. \tag{H1}$$

Оператор объявление.

$$\langle\, \mathcal{HD}ec(Q, DataDecln) \,\rangle\, DataDecln \,\langle\, Q \,\rangle. \tag{A2}$$

Оператор вычисления выражения.

$$\frac{\mathcal{D}om(Expn)}{\langle\, \mathcal{U}pd(Q, Expn) \,\rangle\, Expn \,\langle\, Q \,\rangle}, \tag{H2}$$

где *Expn* — выражение без вызовов функций.

$$\frac{P \Rightarrow P'(\overline{v} \leftarrow \overline{e}) \qquad \langle\, P' \,\rangle\, B \,\langle\, Q' \,\rangle \qquad P \wedge Q'(\overline{v} \leftarrow \overline{e}, \mathrm{f} \leftarrow z) \Rightarrow Q(lval \leftarrow z)}{\langle\, P \,\rangle\, lval = \mathrm{f}(\overline{e});\, \langle\, Q \,\rangle}, \tag{H3}$$

где B — тело функции f, $\overline{v}$ и $\overline{e}$ — списки формальных параметров и фактических аргументов, z — идентификатор ранее не встречавшийся.

Последовательность операторов.

$$\frac{\langle\, P \,\rangle\, A_2 \,\langle\, S \,\rangle \qquad \langle\, S \,\rangle\, A_2 \,\langle\, Q \,\rangle}{\langle\, P \,\rangle\, A_1\, A_2 \,\langle\, Q \,\rangle}. \tag{H4}$$

Блок.

$$\frac{\langle\, P \,\rangle\, A \,\langle\, Q \,\rangle}{\langle\, P \,\rangle\, \{A\} \,\langle\, Q \,\rangle}. \tag{H5}$$

Помеченный оператор.

$$\frac{\langle\, P \,\rangle\, A \,\langle\, Q \,\rangle}{\langle\, P \,\rangle\, \mathrm{L} : A \,\langle\, Q \,\rangle}. \tag{H6}$$

Условный оператор.

$$\frac{\langle\, P \wedge \alpha \,\rangle\, A_1 \,\langle\, Q \,\rangle \qquad \langle\, P \wedge \neg\alpha \,\rangle\, A_2 \,\langle\, Q \,\rangle}{\langle\, P \,\rangle\, \mathsf{if}\ (\alpha)\ A_1\ \mathsf{else}\ A_2 \,\langle\, Q \,\rangle}. \tag{H7}$$

Оператор переключатель.

$$\frac{\langle\, P \wedge (z = k_i) \,\rangle\, A_i \,\langle\, Q \,\rangle \qquad \langle\, P \wedge (z \notin \{\ldots k_i \ldots\}) \,\rangle\, B \,\langle\, Q \,\rangle}{\langle\, P \,\rangle\, \mathsf{switch}\ (z)\{\ldots \mathsf{case}\ k_i : A_i \ldots \mathsf{default} : B\} \,\langle\, Q \,\rangle}. \tag{H8}$$

Цикл while.

$$\frac{\langle\, INV \wedge \alpha \,\rangle\, A \,\langle\, INV \,\rangle}{\langle\, INV \,\rangle\, \mathsf{while}\ (\alpha)\ A \,\langle\, INV \wedge \neg\alpha \,\rangle}. \tag{H9}$$

Передача управления.

$$\frac{P \Rightarrow INV}{\langle\, P \,\rangle\, \mathsf{goto}\ \mathrm{L} \,\langle\, \mathrm{L} : INV \,\rangle}. \tag{H10}$$

Процедуры и функция main.

$$\frac{\langle\, P' \,\rangle\, B \,\langle\, Q' \,\rangle \qquad P \Rightarrow P'(\overline{p}, \overline{e}) \wedge \mathcal{F}ree(Q'(\overline{p}, \overline{e}) \Rightarrow Q(\overline{p}, \overline{e}))}{\langle\, P \,\rangle\, \mathrm{f}(\overline{p}, \overline{e}) \,\langle\, Q \,\rangle}. \tag{H11}$$

$$\frac{\langle\, P' \,\rangle\, B \,\langle\, Q' \,\rangle \qquad P \Rightarrow P'(\overline{p}, \overline{e}) \wedge \mathcal{F}ree(Q'(\overline{p}, \overline{e}) \Rightarrow Q(\overline{p}, \overline{e}))}{\langle\, P \,\rangle\, \mathrm{f}(\overline{p}, \overline{e}) \,\langle\, Q \,\rangle}, \tag{H12}$$

при $\mathsf{void}\ \mathrm{f}(\overline{x}, \overline{v})B(\mathrm{f})$ и $\langle\, P' \,\rangle\, \mathrm{f}(\overline{x}, \overline{v}) \,\langle\, Q' \,\rangle$.

$$\frac{\langle\, P \,\rangle\, B \,\langle\, Q \,\rangle}{\langle\, P \,\rangle\, \mathrm{ret\_type\ main}()B \,\langle\, Q \,\rangle}. \tag{H13}$$

Здесь $\overline{x}$ — список формальных параметров-указателей, $\overline{v}$ — список формальных параметров, не являющихся указателями, $\overline{p}$ — список фактических аргументов-указателей, $\overline{e}$ — список фактических аргументов, не

являющихся указателями, B — тело процедуры f, $\mathcal{F}ree(S)$ — есть выражение S, в котором все переменные, являющиеся ссылочными классами, заменены на уникальные идентификаторы, ранее не встречавшиеся.

Как и в обычных логических системах вывода, в аксиоматической системе HSC в качестве базиса должны быть заданы некоторые истинные утверждения — аксиомы. Они используются в качестве посылок в правиле усиления посылки/ослабления заключения (H1). Набор этих аксиом зависит от класса программ, для которых проводится верификация, т. е. от проблемной области. В качестве наиболее распространенной базы можно задать все истинные утверждения исчисления предикатов первого порядка над массивами.

Как видно, аксиоматическая семантика для языка C-light-kernel получилась предельно простой. В этом и заключалась основная цель перевода из C-light в C-light-kernel. Поясним аксиомы и правила системы HSC подробнее.

Пустой оператор. Самым простым оператором является пустой. Он не производит никаких действий, т. е. не изменяет содержимого памяти, поэтому аксиома (A1) для него очевидна.

Усиление посылки/ослабление заключения. Также интуитивно понятным является закон консеквенции (логического следования) или правило для усиления предусловия и ослабления заключения (H1), без которого вывод был бы невозможен. В частности оно позволяет заменять пред- и постусловие эквивалентными утверждениями. В качестве следствий этого правила можно получить тройки Хоара

$$\langle\, P \,\rangle\, A \,\langle\, true \,\rangle \qquad \text{и} \qquad \langle\, false \,\rangle\, A \,\langle\, Q \,\rangle,$$

истинные для любых P, Q, A. Следуя названию правила, *true* — слабейшее постусловие, *false* — сильнейшее предусловие.

Наибольший интерес представляют слабейшее предусловие и сильнейшее постусловие. Слабейшее предусловие P^* для фиксированных постусловия Q и оператора A удовлетворяет соотношениям

$$\langle\, P^* \,\rangle\, A \,\langle\, Q \,\rangle \qquad \text{и} \qquad \forall P(\langle\, P \,\rangle\, A \,\langle\, Q \,\rangle \vdash P \Rightarrow P^*),$$

а сильнейшее постусловие Q^* для фиксированных предусловия P и оператора A удовлетворяет соотношениям

$$\langle\, P \,\rangle\, A \,\langle\, Q^* \,\rangle \qquad \text{и} \qquad \forall Q(\langle\, P \,\rangle\, A \,\langle\, Q \,\rangle \vdash Q^* \Rightarrow Q).$$

Слабейшее предусловие P^* полностью характеризует множество исходных данных для заданных Q и A, а сильнейшее постусловие Q^* — полная характеристика функции, вычисляемой A для области исходных данных, определяемой P.

Объявления. Как и в случае операционной семантики, чтобы не задавать отдельные аксиомы для разнообразных случаев деклараций, аксиома (A2) задается в наиболее общем случае, а разбор случаев и изменение предусловия возложены на рекурсивную функцию $\mathcal{HD}ec$. В случае переменных она является аналогом одноименной функции из операционной семантики с той лишь разницей, что здесь первым аргументом является не состояние, а логическая формула (т. е. фактически множество состояний). Также в этой функции рассматриваются объявления типов и функций. Как и ранее рассмотрим функцию $\mathcal{HD}ec$ на примере целочисленных переменных, что не ограничивает общности.

$\mathcal{HD}ec$

$(Q, \mathsf{static\ int}\ v;) \quad = \quad \mathrm{int}(v) \Rightarrow Q(v \leftarrow 0);$

$(Q, \mathsf{auto\ int}\ v;) \quad = \quad \mathrm{int}(v) \Rightarrow Q;$

$(Q, \mathsf{int}\ v = exp;) \quad = \quad \mathrm{int}(v) \Rightarrow Q(v \leftarrow exp),$

где int() — предикат, означающий, что переменная имеет тип int. Первые 3 ветви определяют обработку объявлений глобальных и локальных простых переменных с инициализацией и без.

$\mathcal{HD}ec$

$(Q, \mathsf{static}\ \mathbf{T}\ \mathrm{A[n]};) \quad = (\mathbf{T}*)(\mathrm{A}) \Rightarrow Q(\mathrm{P}\#\mathbf{T} \leftarrow \mathcal{I}nit(\mathrm{P}\#\mathbf{T}, \mathrm{A}, \mathrm{A} + \mathrm{n} - 1, 0));$

$(Q, \mathsf{auto}\ \mathbf{T}\ \mathrm{A[n]};) \quad = (\mathbf{T}*)(\mathrm{A}) \Rightarrow Q;$

$(Q, \mathbf{T}\ \mathrm{A[n]} = \{l_0, ..., l_k\}) \quad = (\mathbf{T}*)(\mathrm{A}) \Rightarrow Q(\mathrm{P}\#\mathbf{T} \leftarrow \langle \mathrm{P}\#\mathbf{T}, \mathrm{A} + i, l_i \rangle)$
$(\mathrm{P}\#\mathbf{T} \leftarrow \mathcal{I}nit(\mathrm{P}\#\mathbf{T}, \mathrm{A} + k + 1, \mathrm{A} + \mathrm{n} - 1, 0)$
при $0 \le i \le k$;

$(Q, \mathsf{struct\ TAG}\ \{fields\ decls\};) \quad = \quad (\mathrm{TAG}{=}\{fields\ decls\}) \Rightarrow Q;$

$(Q, \mathsf{struct\ TAG}\ v;) \quad = \quad \mathrm{TAG}(v) \Rightarrow Q(\forall v.x -$
$(\mathrm{v} \leftarrow \langle v, x, init\_val \rangle)).$

Здесь приведены ветви для обработки объявлений указателей, массивов и структур. Заметим, что идентификатор массива преобразуется к указателю на первый элемент. Поскольку после входного анализатора объявления и инициализации будут разделены, то правила для объявлений с одновременной инициализацией излишни.

Необходимо добавить ветвь для typedef декларации.

$\mathcal{HD}ec$

$(Q,$ typedef type_spec IdLexem$)$ $=$ $(\text{IdLexem} = \text{type\_spec}) \Rightarrow Q.$

Оператор-выражение. По аналогии с декларациями, правило (H2) для простого оператора-выражения тоже задается в наиболее общем виде. Функция $\mathcal{U}pd$ производит анализ выражения и соответствующим образом изменяет предусловие. Также в отличие от классических работ есть посылка $\mathcal{D}om(exp)$.

$\mathcal{U}pd$

$(Q, \text{A}[i] = rval;)$ $=$ $(\mathbf{T}*)(\text{A}) \Rightarrow Q(\text{P}\#\mathbf{T} \leftarrow \langle \text{P}\#\mathbf{T}, \text{A} + i, rval\rangle);$

$(Q, s.f = rval;)$ $=$ $Q(s \leftarrow \langle s, f, rval\rangle);$

$(Q, lval = rval;)$ $=$ $Q(lval \leftarrow rval)$ в остальных случаях.

Случаи соответствующие работе с указателями подробно рассмотрены в следующем разделе.

Как и в операционной семантике, присваивание с вызовом функции (H3) разбивается на вызов и присваивание возвращаемого значения. Возвращаемое значение обозначается тем же идентификатором, что и функция. Связь этого значения с параметрами определяется из спецификаций тела функции. Отметим, что подстановка нового уникального идентификатора z вместо идентификатора f является просто переименованием возвращаемого значения, но это необходимо, иначе в полученных условиях корректности невозможно будет различить результаты вызовов функции с разными аргументами. Это правило не является просто композицией правил (H1), (H2) и (H11), так как в третьей посылке добавляется конъюнкция с предусловием. Без нее была бы потеряна связь с исполнением программы, предшествующим вызову. В обзоре выражений языка отмечалось, что в связи с этим правилом было сформулировано ограничение: функции, возвращающие значения, не изменяют внешних переменных и не имеют параметров указателей. В противном случае функции необходимо преобразовывать в процедуры с дополнительными параметрами.

Композиция операторов. Самый простой способ композиции операторов — это последовательность. Ее семантика характеризуется тем фактом, что свойства в неизменной форме могут переноситься из постусловия предшествующего оператора в предусловие последующего в соответствии с правилом (H4). Следует отметить, что в отличие от работ [3, 5], пара операторов в заключении правила не разделяется точкой с запятой, так как в C этот разделитель используется для указания конца оператора, а не для композиции. Также нет необходимости обобщать это правило на случай группы операторов, поскольку блок сам является оператором и правило (H5) просто убирает фигурные скобки, переходя к содержимому блока.

Метки. С помощью оператора goto можно передавать управление как вперед, так и назад. Если управление передается назад, то в программе появляется цикл и нужен инвариант (см. далее). Как уже отмечалось, специально выделенных мест для записи инвариантов циклов нет и инвариантом является первая аннотация в теле цикла. Поэтому единое для обоих случаев правило (H6) просто снимает метку. Если в начале стояла аннотация-инвариант, то потом по аксиоме для пустого оператора она попадет в посылку условия корректности для данного линейного участка.

Условные операторы. Правило вывода для условного оператора if (H7) естественным образом отражает его альтернативные действия. В качестве обобщения этого правила получается правило (H8) для оператора switch.

Цикл. В отличие от всех конструкций, рассмотренных до сих пор, цикл не задает последовательность операторов фиксированной длины. Число итераций может зависеть от начальных данных. Следствием этого является утверждение (e) в лемме 6, где семантика цикла while определена как бесконечное объединение. Очевидно, что правило вывода с бесконечным числом посылок для верификации не представляет интереса. Более того, в аксиоматической семантике в отличие от операционной не моделируется реальное исполнение программы. Поэтому для определения свойств цикла используется принцип индукции, т. е. предполагается, что на каждой итерации выполняется некоторое инвариантное свойство. Инвариантом для цикла while (α) A называется предикат INV такой, что выполнены два свойства:

a) он истинен на входе цикла,

b) истинна тройка Хоара $\langle\,\mathrm{INV} \wedge \alpha\,\rangle\, A\, \langle\,\mathrm{INV}\,\rangle$. Т. е. любая итерация сохраняет это свойство.

Из определения цикла while следует, что постусловием должно быть утверждение INV$\wedge\neg\alpha$. Тогда правило системы HSC для цикла примет вид (H9).

Передача управления. Правило вывода для оператора goto (H10) отражает определение функции $\mathcal{L}abel$ из операционной семантики. Если в теле программы есть фрагмент, начинающийся с оператора, помеченного меткой L, то управление передается на него. При этом вне зависимости от того, куда передается управление (вперед или назад), рассматривается некоторый инвариант L:INV. Если это цикл, то L:INV — инвариант этого цикла, а если управление передается вперед, то формула, полученная при выводе соответствующего фрагмента. Здесь существенно ограничение языка на оператор goto. Без него правило выглядело бы очень сложно.

Процедуры и функция main. Вообще говоря, в языке C процедуры отдельно не выделяются, это функции с типом возвращаемого значения void. Обработка тел функций и процедур происходит одинаково. Процедуры используются в основном для изменения переменных, передаваемых по ссылке (через указатели). Это изменение должно быть явным образом отражено в аннотациях. Поэтому при вызове процедуры происходит поиск спецификации процедуры. Очевидно, что спецификацией процедуры являются предусловие и постусловие ее тела. Таким образом, посылкой правила для нерекурсивной процедуры (H11) является тройка Хоара для тела B. Для того чтобы связать действие процедуры, описанное в ее спецификации, с пред- и постусловиями, соответствующими точке вызова, используется вторая посылка. Самое важное свойство этой посылки состоит в том, что в последней импликации все ссылочные классы переименованы уникальным образом (выражение $\mathcal{F}ree(...)$). Действие этого переименовывания совершенно аналогично связыванию всех ссылочных классов кванторами всеобщности. Это означает, что связь постусловий Q и Q' не зависит от изменений объектов, переданных через указатели, в теле процедуры, т. е. эта связь сохраняется при любом вызове.

В случае рекурсивной процедуры существует та же проблема, что и в цикле while. Поскольку реальное число срабатываний неизвестно, то

используется индуктивное предположение (гипотеза) о том, что внутренний вызов (в теле самой процедуры) удовлетворяет спецификации тела, а в остальном правило (H12) совершенно аналогично правилу для нерекурсивной процедуры.

Правило (H13) обрабатывает определение функции main. Поскольку эта функция является основной в программе, то правило просто объявляет пред- и постусловия программы пред- и постусловиями функции main.

4.2. Аксиоматическая семантика указателей

Указатели представляют собой значительную трудность для формальной верификации особенно в таком языке, как C. Связано это в первую очередь с низким уровнем понятия «указатель». Поэтому семантика указателей получается либо слишком громоздкой, либо не позволяет доказывать все свойства, относящиеся к указателям. Ряд ограничений на язык C, сформулированных при определении языков C-Light и C-Light-kernel, относится к указателям. Напомним их:

- все указатели 32-битные; модификаторы `near`, `far` и `huge` игнорируются;
- указатели на функции не поддерживаются;
- используется плоская (flat) модель памяти;
- приведение типа (T)ptr допустимо только если ptr имеет тип `void`;
- для выделения и освобождения памяти используются операции `new` и `delete` из языка C++, операция взятия адреса запрещена.

Идентификаторы ссылочных классов. Как отмечалось в работе [11], основная проблема указателей языка Паскаль (аналогично в языке C-light-kernel) состоит в том, что у динамически создаваемых объектов нет имен, поэтому невозможно делать подстановки в аннотациях. В качестве выхода было предложено ввести в язык спецификаций новые переменные — идентификаторы ссылочных классов, и соответственно новый составной тип — ссылочный класс. В языке C-light-kernel они, как правило, имеют вид P#<id>, где <id> есть любой допустимый идентификатор типа. Неформально, если тип T0 объявлен как typedef T* T0, то P#T представляет собой неограниченное множество объектов типа T, на которые могут указывать переменные типа T0. Ссылочные классы не являются элементами языка C-light-kernel, это примитивы языка

спецификаций, которые ведут себя подобно неограниченным массивам. Тип переменной P#T есть *ссылочный класс* типа T, и обозначается он как **Ref**(T). Заметим, что синтаксис P#T, предложенный в [11], удобен тем, что имя переменной содержит информацию о ее типе.

Между ссылочными классами языков Паскаль и C-light-kernel есть принципиальная разница. В Паскале элементами ссылочного класса являются только динамически создаваемые объекты, и любому указателю можно присвоить либо результат операции new, либо значение другого указателя. Но в любом случае это адреса объектов в динамической памяти — «куче». В языке C-light-kernel, как и в C, указателю можно присвоить произвольное значение типа unsigned int. Например, зная адрес конкретного порта, можно организовывать некоторый ввод-вывод. Следовательно, если в языке Паскаль в начале работы программы, пока не создан ни один динамический объект, любой ссылочный класс есть пустое множество, то в C-light-kernel любой ссылочный класс «содержит» всю доступную статическую память. Естественно, что в зависимости от типа T ссылочные классы по разному рассматривают эту память. Так класс P#char разбивает ее на байты, а с точки зрения класса P#int это множество четырехбайтных[2] ячеек. Очевидно, что такая трактовка сразу приводит к проблеме объединений, когда, начиная с одного и того же адреса могут располагаться различные объекты. Т. е., например, необходимо внимательно отслеживать выражения приведения типов. Тем не менее в семантике языка C-light-kernel не рассматривается конкретная архитектура или способ раскладки данных в памяти. В частности, если пользователь работает с конкретными адресами, то в спецификации должно быть отражено предполагаемое содержимое этой памяти.

Как уже упоминалось, операция взятия адреса (&) в языке C-light разрешена, а в C-light-kernel запрещена, поэтому не делается никаких предположений о размещении статических объектов в ссылочных классах. При трансляции используется свойство двойственности операций адресации и разыменовывания. Вместо переменной, на которую делается хотя бы одна ссылка, заводится уникальный указатель.

Каждый ссылочный класс расширяется неопределенным значением — ω.

Функции и предикаты на ссылочных классах. Ссылочные классы ведут себя подобно неограниченным массивам, поэтому синтаксис

[2]это верно только для C-light-kernel.

некоторых операций над классами похож на синтаксис, предложенный в [12]. Введем функциональные символы, соответствующие операциям разыменовывания, присваивания, выделения и освобождения памяти в языке C-Light-kernel. Для спецификации того, что указатель указывает на динамический объект в ссылочном классе, введем специальный предикатный символ PnTo.

выбор:	P#T[*ptr*],
присваивание:	$\langle$P#T, *ptr*, *value*$\rangle$,
расширение:	P#T$\cup\{addr1, addr2\}$, где $addr1 \leq addr2$,
сужение:	P#T$\setminus\{address\}$,
ссылочный предикат:	PnTo(P#T, ptr).

Формальное определение этих операций и предиката задается достаточно обширной системой аксиом, которые используются при верификации.

Заметим, что с использованием операций расширения и сужения становится возможным отлавливать некоторые ошибки, связанные с неправильным распределением памяти. Например, ошибкой является попытка вернуть в кучу память, связанную с указателем, который указывает на статический элемент. Т. е. любой операции сужения должна где-то ранее предшествовать операция расширения для того же указателя. Далее, как известно, игнорирование операции возврата памяти может привести к «утечкам» памяти (memory leaks). Поэтому ситуация, когда для операции расширения нет парной ей операции сужения, может служить предупреждением верификатору.

Переписывание аннотаций и операторов присваивания. При генерации условий корректности для указателей языка C-Light-kernel возникает проблема. Для каждого указателя при операции разыменовывания необходимо помнить его тип, чтобы знать, в каком ссылочном классе производить соответствующие операции. Хранение глобальной информации о типах может сильно усложнить генерацию. Наиболее простое решение — переписывание, которое задается в виде рекурсивной функции Tr:

Tr(V) = V, если V — идентификатор без ссылок на него;
Tr(*x) = P#T[Tr(x)], если x — указатель на тип T;
Tr(a[i]) = P#T[Tr(a)+Tr(i)], если a — массив типа T;
Tr(R.F) = Tr(R).F;
Tr(R->F) = Tr((*R).F).

Эта рекурсивная процедура реализовывана как часть входного анализатора системы. Следует напомнить, что при трансляции происходит еще одно переписывание более глобального характера, когда идентификатор переменной, на которую есть хотя бы одна ссылка, заменяется на разыменованный уникальный указатель, работающий только с динамической памятью.

Отметим, что при использовании переписывания до начала генерации аксиомы семантики для массивов становятся избыточными.

Правила семантики. Из всех правил аксиоматической семантики, имеющих дело с указателями, наибольший интерес представляют правило для присваивания с косвенной адресацией (разыменовывание указателя) и правила для выделения/освобождения памяти. Правило для объявления указателей ничем не отличается от правила для объявления обычной переменной. Заметим, что поскольку частично используется синтаксис языка C++, то выделение памяти имеет вид оператора присваивания.

1. Аксиомы для декларации указателей:

$$\langle (\mathrm{T}*)(\mathrm{ptr}) \Rightarrow \mathrm{Q}(\mathrm{ptr} \leftarrow \mathrm{NULL}) \rangle \ \mathsf{static}\ \mathrm{T}*\ \ \mathrm{ptr}\ \langle \mathrm{Q} \rangle. \tag{26}$$

$$\langle (\mathrm{T}*)(\mathrm{ptr}) \Rightarrow \mathrm{Q} \rangle \ \mathsf{auto}\ \mathrm{T}*\ \ \mathrm{ptr}\ \langle \mathrm{Q} \rangle. \tag{27}$$

$$\langle (\mathrm{T}*)(\mathrm{ptr}) \Rightarrow \mathrm{Q}(\mathrm{ptr} \leftarrow \mathrm{exp}) \rangle\ \mathrm{T}*\ \mathrm{ptr} = \mathrm{exp}\ \langle \mathrm{Q} \rangle, \tag{28}$$

если истинно $\mathcal{D}om(\mathrm{exp})$.

Как и в случае простых переменных, автоматический указатель не инициализируется.

2. Аксиома для присваивания с косвенной адресацией:

$$\langle\ \mathrm{Q}(\mathrm{P}\#\mathrm{T} \leftarrow \langle\ \mathrm{P}\#\mathrm{T}, \mathrm{ptr}, \mathrm{exp}\ \rangle)\ \rangle\ \mathrm{P}\#\mathrm{T}[\mathrm{ptr}] = \mathrm{exp}\ \langle\ \mathrm{Q}\ \rangle, \tag{29}$$

если истинно $\mathcal{D}om(\mathrm{exp})$.

Таким образом производится обновление всего ссылочного класса. Отметим, что нет необходимости отдельно рассматривать присваивание просто указателю, так как с этой позиции указатель ничем не отличается от простой переменной. Также не нужно отслеживать информацию о ссылках на переменные (aliasing), поскольку операция получения адреса запрещена.

3. Аксиомы для выделения памяти под переменную и массив:

$$\langle\, \neg \mathrm{PnTo}(\mathrm{P}\#\mathrm{T}, \mathrm{ptr}') \Rightarrow \mathrm{Q}(\mathrm{ptr} \leftarrow \mathrm{ptr}')(\mathrm{P}\#\mathrm{T} \leftarrow \mathrm{P}\#\mathrm{T} \cup \{\mathrm{ptr}', \mathrm{ptr}'\})\,\rangle \\ \mathrm{ptr} = \mathsf{new}\ \mathrm{T}\,\langle\, \mathrm{Q}\,\rangle. \tag{30}$$

$$\langle\, \neg \mathrm{PnTo}(\mathrm{P}\#\mathrm{T}, \mathrm{ptr}') \Rightarrow \langle\, \mathrm{Q}(\mathrm{ptr} \leftarrow \mathrm{ptr}') \\ \mathrm{P}\#\mathrm{T} \leftarrow \mathrm{P}\#\mathrm{T} \cup \{\mathrm{ptr}', \mathrm{ptr}' + \exp - 1\})\,\rangle\, \mathrm{ptr} = \mathsf{new}\ \mathrm{T}[\exp]\,\langle\, \mathrm{Q}\,\rangle, \tag{31}$$

где ptr' — идентификатор, ранее не встречавшийся.

Как и в операционной семантике, необходимо явно указать, что при динамическом выделении памяти указатель получает уникальное значение — адрес в куче. Однако само это значение известно только при реальном исполнении программы, поэтому вводится новый идентификатор ptr' и в предусловие помещается посылка $\neg\mathrm{PnTo}(\mathrm{P}\#\mathrm{T}, \mathrm{ptr}')$, которая и говорит, что это новое значение не указывает ни на какую ячейку или группу ячеек в куче.

4. Аксиомы для освобождения памяти:

$$\langle\, \mathrm{PnTo}(\mathrm{P}\#\mathrm{T}, \mathrm{ptr}) \Rightarrow \mathrm{Q}(\mathrm{P}\#\mathrm{T} \leftarrow \mathrm{P}\#\mathrm{T} \setminus \{\mathrm{ptr}\})\,\rangle\, \texttt{delete}\ \mathrm{ptr}\,\langle\, \mathrm{Q}\,\rangle. \tag{32}$$

$$\langle\, \mathrm{PnTo}(\mathrm{P}\#\mathrm{T}, \mathrm{ptr}) \Rightarrow \mathrm{Q}(\mathrm{P}\#\mathrm{T} \leftarrow \mathrm{P}\#\mathrm{T} \setminus \{\mathrm{ptr}\})\,\rangle\, \texttt{delete[]}\ \mathrm{ptr}\,\langle\, \mathrm{Q}\,\rangle. \tag{33}$$

Аксиомы очевидным образом отражают тот факт, что освободить можно только ранее выделенную память. Поэтому в предусловиях стоят посылки $\mathrm{PnTo}(\mathrm{P}\#\mathrm{T}, \mathrm{ptr})$. Также аксиомы объясняют, почему в операции сужения ссылочного класса не указывается верхняя граница области памяти: вывод в стандартной аксиоматике Хоара является неоднозначным процессом, и обработка операции delete может происходить до обработки парной ей операции new. Однако никакой неоднозначности в вычислении значений при доказательстве не возникает. Аксиоматическое определение проблемной области указателей, т. е. ссылочных классов и операций над ними, включает корректную обработку освобождения памяти. Более того, и по синтаксису языка C++ размер удаляемого массива не указывается.

5. НЕПРОТИВОРЕЧИВОСТЬ АКСИОМАТИЧЕСКОЙ СЕМАНТИКИ

Каким образом можно обосновать, что доказана истинность тройки Хоара в смысле частичной корректности $\models \langle \mathrm{P} \rangle \mathrm{A} \langle \mathrm{Q} \rangle$? Ведь тройка

Хоара $\langle P \rangle A \langle Q \rangle$ и ее вывод в аксиоматической системе вывода — это всего лишь строки символов, а $\models \langle P \rangle A \langle Q \rangle$ — это утверждение о свойствах программы. Т. е. необходимо интерпретировать вывод тройки как доказательство. Это фундаментальный вопрос, касающийся всех логических систем вывода. Для его разрешения необходимо, чтобы система вывода обладала свойством непротиворечивости, т. е., чтобы при выводе из общезначимых формул получались общезначимые. Обычно вместе со свойством непротиворечивости стараются доказать и полноту системы вывода. Полнота в нашем случае гарантирует, что можно получить все истинные утверждения о свойствах программ. В текущей версии системы свойство полноты было решено не рассматривать. Во-первых, доказательство полноты основано на индуктивном определении слабейшего предусловия и гораздо сложнее, чем доказательство непротиворечивости. Во-вторых, здесь есть принципиальное препятствие в виде неоднозначного соответствия программ и спецификаций (и условий корректности). Как известно, одной аннотированной программе соответствует один набор условий корректности, но одному набору условий корректности соответствует множество программ.

Определение. Правило вывода

$$\frac{\phi_1, ..., \phi_k}{\phi_{k+1}}$$

называется непротиворечивым для частичной корректности, если истинность ϕ_1,..., ϕ_k в смысле частичной корректности влечет истинность ϕ_{k+1} в смысле частичной корректности. (Если какие-то из формул ϕ_i являются утверждениями, то отождествляем их истинность в смысле частичной корректности с истинностью в обычном смысле).

Теорема. Система вывода HSC непротиворечива для свойства частичной корректности для троек языка C-light-kernel. Т. е. $\vdash_{\mathrm{HSC}} \phi$ влечет $\models \phi$.

Доказательство. Из вида системы вывода HSC следует, что достаточно доказать истинность всех аксиом системы HSC в смысле частичной корректности и непротиворечивость всех правил системы HSC для частичной корректности. Тогда результат следует из индукции по высоте дерева вывода. Как следует из определения, непротиворечивость тройки Хоара означает, что семантика частичной корректности программы, примененная ко множеству состояний, в которых истинно

предусловие, есть подмножество множества состояний, в которых истинно постусловие.

Пустой оператор. Очевидно, что $\mathcal{M}[\![;]\!](\|P\|) = \|P\|$ для любого утверждения P, следовательно аксиома (A1) истинна в смысле частичной корректности.

Усиление посылки/ослабление заключения. Предположим, что

$$P \Rightarrow S, \qquad \mathcal{M}[\![A]\!](\|S\|) \subseteq \|T\| \qquad \text{и} \quad T \Rightarrow Q.$$

Тогда по лемме 2(4) $\|P\| \subseteq \|S\|$и$\|T\| \subseteq \|Q\|$, поэтому по свойству монотонности семантики частичной корректности (лемма 5(a))

$$\mathcal{M}[\![A]\!](\|P\|) \subseteq \mathcal{M}[\![A]\!](\|S\|) \subseteq \|T\| \subseteq \|Q\|,$$

т. е. правило (H1) непротиворечиво для частичной корректности.

Оператор объявление. В доказательстве рассматриваются соответствующие ветви функций $\mathcal{D}ec$ и $\mathcal{HD}ec$. Как и в определении этих функций доказательство для деклараций простых переменных проводится для случая типа int.

a) Типы. Для typedef-декларации аксиома (A2) примет вид

$$\langle\, (\text{IdLexem} = \text{type\_spec}) \Rightarrow P \,\rangle\, \textsf{typedef}\ \text{type\_spec IdLexem}\, \langle\, P \,\rangle.$$

Пусть P — некоторое утверждение. Тогда по лемме о подстановке, если $\mathcal{M}[\![\textsf{typedef}\ \text{type\_spec IdLexem}]\!](\sigma) = \{\tau\}$, то

$$\sigma \models P \quad \text{iff} \quad \tau \models P,$$

если IdLexem — синоним типа type_spec, что эквивалентно

$$\sigma \models (\text{IdLexem} = \text{type\_spec} \Rightarrow P) \quad \text{iff} \quad \tau \models P.$$

Отсюда $\mathcal{M}[\![\textsf{typedef}\ \text{type\_spec IdLexem}]\!](\|\text{IdLexem} = \text{type\_spec} \Rightarrow P\|) \subseteq \|P\|$.

b) Переменные. Если объявление имеет вид `static int v;`, то аксиома (8) примет вид $\langle \textsf{static int}\ v, \sigma\rangle \rightarrow \langle E, \sigma(\mathrm{M}(v \leftarrow 0), \Gamma(v \leftarrow \textsf{int}))\rangle$, а аксиома (A2), соответственно — $\langle\, \mathrm{int}(v) \Rightarrow P(v \leftarrow 0) \,\rangle\, \textsf{static int}\ v\, \langle\, P \,\rangle$.

Пусть P — некоторое утверждение. По лемме о подстановке, если $\mathcal{M}[\![\mathsf{static\ int}\ v]\!](\sigma) = \{\tau\}$, то

$$\sigma \models P(v \leftarrow 0) \quad \text{iff} \quad \tau \models P,$$

если тип переменной v есть int, что эквивалентно

$$\sigma \models \mathrm{int}(v) \Rightarrow P(v \leftarrow 0) \quad \text{iff} \quad \tau \models P.$$

Отсюда $\mathcal{M}[\![\mathsf{static\ int}\ v]\!](\|\mathrm{int}(v) \Rightarrow P(v \leftarrow 0)\|) \subseteq \|P\|$. Случаи декларации автоматической переменной и декларации с инициализацией доказываются аналогично.

Доказательство для деклараций структур проводится аналогично на основе леммы о подстановке, т. е. происходят подстановки термов вида $\mathrm{upd}\langle ... \rangle$ вместо идентификаторов структурных переменных с учетом того, что объявление структуры вводит новый тип.

Таким образом, для любой декларации *DataDecln*

$$\mathcal{M}[\![DataDecln]\!](\|\mathcal{HD}ec(P, DataDecln)\|) \subseteq \|P\|,$$

т. е. аксиома (A2) истинна в смысле частичной корректности.

Оператор выражение. Как уже отмечалось, в языке C-light-kernel операторами-выражениями являются только оператор присваивания и вызов процедуры. В правилах операционной и аксиоматической семантик используются функции $\mathcal{U}pd$, которые определяют результат выполнения этого оператора для различных правых частей, а также отслеживают возможный вызов операции new в правой части. Вызовы процедур рассмотрены отдельно.

Доказательство аксиом для присваиваний различным объектам основано на лемме о подстановке, в которой разобраны все случаи. Поэтому подробное доказательство проводится для присваивания простой переменной и указателю с косвенной адресацией, а все остальные объекты (массивы, записи) по аналогии. Также в каждом случае предполагается, что истинен предикат $\mathcal{D}om$ от правой части *rval*, поэтому для краткости эта посылка не пишется. В случае ложного значения в операционной семантике происходит переход в состояние fail и остановка исполнения программы, а в аксиоматической семантике это означает прекращение вывода, что полностью эквивалентно.

a) Для простого присваивания (без вызова функции) аксиома примет вид $\langle\, P(lval \leftarrow rval)\,\rangle\, lval = rval\, \langle\, P\,\rangle$, а аксиома (11) операционной семантики соответственно примет вид

$$\langle lval = rval, \sigma\rangle \to \langle rval, \sigma(\mathrm{M}(lval \leftarrow rval))\rangle. \qquad (*)$$

Пусть P — некоторое утверждение. По лемме о подстановке и аксиоме $(*)$, если $\mathcal{M}[\![lval = rval]\!](\sigma) = \{\tau\}$, то

$$\sigma \models P(lval \leftarrow rval) \quad \text{iff} \quad \tau \models P.$$

Отсюда $\mathcal{M}[\![lval = rval]\!](\|P(lval \leftarrow rval)\|) \subseteq \|P\|$, т. е. аксиома (H2) для случая присваивания простой переменной истинна в смысле частичной корректности.

b) Присваивание разыменованному указателю. В этом случае аксиома (11) после переписывания (*ptr $\mapsto$ P#T[ptr]) примет вид

$$\langle \mathrm{P\#T[ptr]} = rval, \sigma\rangle \to \langle rval, \sigma(\mathrm{M}(\mathrm{P\#T} \leftarrow \langle \mathrm{P\#T}, \mathrm{ptr}, rval\rangle))\rangle. \qquad (*)$$

Отметим, что переписывание избавляет от небходимости проверять тип указателя в аксиоматической семантике. Пусть Q — некоторое утверждение. По лемме о подстановке и $(*)$, если $\mathcal{M}[\![\mathrm{P\#T[ptr]} = rval]\!](\sigma) = \{\tau\}$, то

$$\sigma \models Q(\mathrm{P\#T} \leftarrow \langle \mathrm{P\#T}, \mathrm{ptr}, rval\rangle) \quad \text{iff} \quad \tau \models Q.$$

Отсюда $\mathcal{M}[\![\mathrm{P\#T[ptr]} = rval]\!](\|Q(\mathrm{P\#T} \leftarrow \langle \mathrm{P\#T}, \mathrm{ptr}, rval\rangle)\|) \subseteq \|Q\|$, т. е. аксиома (29) истинна в смысле частичной корректности.

c) Присваивание с вызовом функции. Предположим, что

$$P \Rightarrow P'(v \leftarrow e), \qquad \mathcal{M}[\![B]\!](\|P'\|) \subseteq \|Q'\| \quad \text{и}$$
$$P \wedge Q'(v \leftarrow e, f \leftarrow z) \Rightarrow Q(lval \leftarrow z).$$

Тогда по лемме 2(4)

$$\|P\| \subseteq \|P(v \leftarrow e)\|, \qquad \|P \wedge Q'(v \leftarrow e, f \leftarrow z)\| \subseteq \|Q(lval \leftarrow z)\|.$$

Учитывая то, что аргументы в языке C-light-kernel, как и в C, передаются по значению, а измененние глобальных переменных в телах функций без передачи их адресов запрещено, получим

$$\mathcal{M}[\![B]\!](\|P'(v \leftarrow e)\|) \subseteq \|Q'(v \leftarrow e)\|.$$

Для изображения возвращаемого значения используется неинтепретируемая константа f (или rezult в операционной семантике), поэтому замена $Q(\mathrm{f} \leftarrow z)$, где z — новая неинтепретируемая константа, есть просто переименование и не меняет смысла утверждения, т. е.

$$\|Q(v \leftarrow e)\| = \|Q(v \leftarrow e, \mathrm{f} \leftarrow z)\|.$$

По определению $\mathcal{M}[\![lval = \mathrm{f}(e)]\!](\sigma) = \mathcal{M}[\![B(v \leftarrow e);\ lval = z]\!](\sigma)$. Как было доказано в случае (a), $\mathcal{M}[\![lval = z]\!](\|Q(lval \leftarrow z)\|) \subseteq \|Q\|$.

Таким образом, используя свойство монотонности, получаем, что $\mathcal{M}[\![lval = \mathrm{f}(e)]\!]\|P\| \subseteq \|Q\|$, т. е. правило (H3) непротиворечиво для частичной корректности.

d) Выделение динамической памяти. Для выделения динамической памяти простой переменной аксиома операционной семантики примет вид

$$\langle \mathrm{ptr} = \mathsf{new}\ \mathrm{T}, \sigma\rangle \rightarrow \langle val, \sigma(\mathrm{M}(\mathrm{ptr} \leftarrow val)(\mathrm{P}\#\mathrm{T} \leftarrow \mathrm{M}_\sigma(\mathrm{P}\#\mathrm{T} \cup \{val\})))\rangle,$$

при $\sigma \not\models \mathrm{PnTo}(\mathrm{P}\#\mathrm{T}, val)$. Пусть Q — некоторое утверждение. По лемме о подстановке и аксиоме, если $\mathcal{M}[\![\mathrm{ptr} = \mathsf{new}\ \mathrm{T}]\!](\sigma) = \{\tau\}$, то

$$\sigma \models \neg\mathrm{PnTo}(\mathrm{P}\#\mathrm{T}, val) \Rightarrow Q(\mathrm{ptr} \leftarrow val)(\mathrm{P}\#\mathrm{T} \leftarrow \mathrm{M}_\sigma(\mathrm{P}\#\mathrm{T} \cup \{val\})) \quad \text{iff } \tau \models Q.$$

Отсюда

$$\mathcal{M}[\![\mathrm{ptr} = \mathsf{new}\ \mathrm{T}]\!](\|\neg\mathrm{PnTo}(\mathrm{P}\#\mathrm{T}, val) \Rightarrow Q(\mathrm{ptr} \leftarrow val)(\mathrm{P}\#\mathrm{T} \leftarrow \mathrm{M}_\sigma(\mathrm{P}\#\mathrm{T} \cup \{val\}))\|) \subseteq \|Q\|.$$

Заменяя val на ptr', получим, что аксиома (30) истинна в смысле частичной корректности.

Доказательство для аксиомы (31) отличается только тем, что в операции расширения ссылочного класса добавляется верхняя граница.

e) Освобождение динамической памяти. Доказательство проводится только для указателя на простую переменную, поскольку для массива аскиома точно такая же. Пусть Q — некоторое утверждение. По лемме о подстановке и аксиоме (13), если $\mathcal{M}[\![\mathsf{delete}\ \mathrm{ptr}]\!](\sigma) = \{\tau\}$, то

$$\sigma \models Q(\mathrm{P}\#\mathrm{T} \leftarrow \mathrm{P}\#\mathrm{T} \setminus \{\mathrm{ptr}\}) \quad \text{iff} \quad \tau \models Q,$$

при PnTo(P#T, ptr) или

$$\sigma \models (\mathrm{PnTo}(\mathrm{P\#T}, \mathrm{ptr}) \Rightarrow Q(\mathrm{P\#T} \leftarrow \mathrm{P\#T} \setminus \{\mathrm{ptr}\})) \quad \text{iff} \quad \tau \models Q.$$

Отсюда $\mathcal{M}[\![\mathsf{delete}\ \mathrm{ptr}]\!](\|\mathrm{PnTo}(\mathrm{P\#T}, \mathrm{ptr}) \Rightarrow Q(\mathrm{P\#T} \leftarrow \mathrm{P\#T} \setminus \{\mathrm{ptr}\})\|) \subseteq \|Q\|$, т. е. аксиомы (32) и (33) истинны в смысле частичной корректности.

Последовательность операторов. Предположим что

$$\mathcal{M}[\![A_1]\!](\|P\|) \subseteq \|S\| \qquad \text{и} \qquad \mathcal{M}[\![A_2]\!](\|S\|) \subseteq \|Q\|.$$

Тогда по свойству монотонности семантики (лемма 5(a)) следует, что

$$\mathcal{M}[\![A_2]\!](\mathcal{M}[\![A_1]\!](\|P\|)) \subseteq \mathcal{M}[\![A_2]\!](\|S\|) \subseteq \|Q\|.$$

Но по лемме 5(b)

$$\mathcal{M}[\![A_1\ A_2]\!](\|P\|) = \mathcal{M}[\![A_2]\!](\mathcal{M}[\![A_1]\!](\|P\|)),$$

поэтому

$$\mathcal{M}[\![A_1\ A_2]\!](\|P\|) \subseteq \|Q\|,$$

т. е. правило (H4) непротиворечиво для частичной корректности.

Блок. Очевидно, что правило (H5) для блока непротиворечиво для частичной корректности, поскольку операционная и аксиоматическая семантика описывают работу с блоком совершенно одинаково — снятие фигурных скобок и переход к содержимому.

Помеченный оператор. Предположим, что

$$\mathcal{M}[\![A]\!](\|P\|) \subseteq \|Q\|.$$

Согласно операционной семантике $\mathcal{M}[\![\mathrm{L} : A]\!](\|P\|) = \mathcal{M}[\![A]\!](\|P\|)$, значит

$$\mathcal{M}[\![\mathrm{L} : A]\!](\|P\|) \subseteq \|Q\|,$$

т. е. правило (H6) непротиворечиво для частичной корректности.

Условный оператор. Предположим что

$$\mathcal{M}[\![A_1]\!](\|P \wedge \alpha\|) \subseteq \|Q\| \qquad \text{и} \qquad \mathcal{M}[\![A_2]\!](\|P \wedge \neg\alpha\|) \subseteq \|Q\|.$$

По лемме 5(c)

$$\mathcal{M}[\![\mathsf{if}(\alpha)\ A_1\ \mathsf{else}\ A_2]\!](\|P\|) = \mathcal{M}[\![A_1]\!](\|P \wedge \alpha\|) \cup \mathcal{M}[\![A_2]\!](\|P \wedge \neg\alpha\|).$$

Тогда

$$\mathcal{M}[\![\mathsf{if}(\alpha)\ A_1\ \mathsf{else}\ A_2]\!](\|P\|) \subseteq \|Q\|,$$

т. е. правило (H7) непротиворечиво для частичной корректности.

Оператор переключатель. Предположим что

$$\mathcal{M}[\![A_i]\!](\|P \wedge (\alpha = k_i)\|) \subseteq \|Q\| \text{ при } i = 0, ..., n \text{ и}$$

$$\mathrm{M}[\![B]\!](\|P \wedge (\alpha \notin \{... k_i ...\})\|) \subseteq \|Q\|.$$

По лемме 5(d)

$$\mathcal{M}[\![\mathsf{switch}(\alpha)\{...;\ \mathsf{case}\ k_i : A_i\ ...;\ \mathsf{default} : B\}]\!](\|P\|) =$$
$$\bigcup_i \mathcal{M}[\![A_i]\!](\|P \wedge (\alpha = k_i)\|) \cup \mathcal{M}[\![B]\!](\|P \wedge (\alpha \notin \{... k_i ...\})\|).$$

Тогда

$$\mathcal{M}[\![\mathsf{switch}(\alpha)\{...;\ \mathsf{case}\ k_i : A_i\ ...;\ \mathsf{default} : B\}]\!](\|P\|) \subseteq \|Q\|,$$

т. е. правило (H8) непротиворечиво для частичной корректности.

Цикл while. Предположим, что для некоторого утверждения P

$$\mathcal{M}[\![A]\!](\|P \wedge \alpha\|) \subseteq \|P\|. \qquad (*)$$

Докажем индукцией по $k \geq 0$, что

$$\mathcal{M}[\![(\mathsf{while}(\alpha)\ A)^k)]\!](\|P\|) \subseteq \|P \wedge \neg\alpha\|.$$

Для $k = 0$ очевидно. Предположим, что $(*)$ выполнено для некоторого $k > 0$. Тогда

$$\begin{aligned}
&\mathcal{M}[\![(\mathsf{while}(\alpha)\ A)^{k+1})]\!](\|P\|) \\
&= (\text{по определению } (\mathsf{while}(\alpha)\ A)^{k+1}) \\
&\quad \mathcal{M}[\![\mathsf{if}(\alpha)\{A; (\mathsf{while}(\alpha)\ A)^k\ \mathsf{else;}\}]\!](\|P\|) \\
&= (\text{лемма 5(c)}) \\
&\quad \mathcal{M}[\![A; (\mathsf{while}(\alpha)\ A)^k]\!](\|P \wedge \alpha\|) \cup \mathcal{M}[\![;]\!](\|P \wedge \neg\alpha\|) \\
&= (\text{лемма 5(b) и семантика пустого оператора}) \\
&\quad \mathcal{M}[\![(\mathsf{while}(\alpha)\ A)^k)]\!](\mathcal{M}[\![A]\!](\|P \wedge \alpha\|)) \cup \|P \wedge \neg\alpha\| \\
&\subseteq (\text{лемма 5(a) и } (*)) \\
&\quad \mathcal{M}[\![(\mathsf{while}(\alpha)\ A)^k)]\!](\|P\|) \cup \|P \wedge \neg\alpha\| \\
&\subseteq (\text{предположение индукции}) \\
&\quad \|P \wedge \neg\alpha\|.
\end{aligned}$$

Таким образом

$$\bigcup_{k=0}^{\infty} \mathcal{M}[\![(\mathsf{while}(\alpha)\ A)^k)]\!](\|P\|) \subseteq \|P \wedge \neg\alpha\|.$$

А по лемме 5(e) получим

$$\mathcal{M}[\![\mathsf{while}(\alpha)\ A]\!](\|P\|) \subseteq \|P \wedge \neg\alpha\|.$$

Заменяя P на INV, получим, что правило (H9) непротиворечиво для частичной корректности.

Передача управления. Предположим, что

$$P \Rightarrow INV.$$

Очевидно, что $\mathcal{M}[\![\mathsf{goto\ L}]\!](\|P\|) = \|P\|$, поскольку передача не изменяет состояния[3]. Тогда по лемме 2(4)

$$\mathcal{M}[\![\mathsf{goto\ L}]\!](\|P\|) \subseteq \|INV\|,$$

следовательно правило (H10) непротиворечиво для частичной корректности.

[3]Как уже отмечалось, это верно только при ограничениях на оператор goto.

Процедуры. a) Нерекурсивная процедура. Предположим, что

$$\mathcal{M}[\![B]\!](\|P'\|) \subseteq \|Q'\| \quad \text{и} \quad P \Rightarrow P'(\overline{p},\overline{e}) \wedge \mathcal{F}ree(Q'(\overline{p},\overline{e}) \Rightarrow Q(\overline{p},\overline{e})).$$

Тогда по лемме 2(4) $\|P\| \subseteq \|P'(\overline{p},\overline{e})\|$ и $\mathcal{F}ree(\|Q'(\overline{p},\overline{e})\| \subseteq \|Q(\overline{p},\overline{e})\|)$, а также $\mathcal{M}[\![B]\!](\|P'(\overline{p},\overline{e})\|) \subseteq \|Q'(\overline{p},\overline{e})\|$, поскольку нет неявных изменений глобальных переменных. Согласно операционной семантике

$$\mathcal{M}[\![\mathrm{f}(\overline{p},\overline{e})]\!](\sigma) = \mathcal{M}[\![B(\overline{p},\overline{e})]\!](\sigma).$$

Действие функции $\mathcal{F}ree()$ есть переименование ссылочных классов эквивалентное связыванию их квантором $\forall$. Это означает, что для любого вызова процедуры f, после ее исполнения из постусловия Q' следует постусловие Q.

Тогда, объединяя по свойству монотонности всю цепочку вложений, получим

$$\mathcal{M}[\![\mathrm{f}(\overline{p},\overline{e})]\!](\|P\|) \subseteq \|Q\|,$$

т. е. правило (H11) непротиворечиво для частичной корректности.

b) Доказательство для правила (H12) аналогично случаю нерекурсивной процедуры. Заметим, что в операционной семантике не делается никаких индуктивных предположений (как для циклов, так и для процедур), поскольку моделируется реальное исполнение. Т. е. правило будет срабатывать столько, сколько будет реальных вызовов. Гипотеза о соответствии внутреннего вызова процедуры спецификации ее тела не является посылкой правила, она передается в соответствующее условие корректности.

c) Правило для определения функции main в точности соответствует операционной семантике. Просходит переход к телу в том же самом состоянии, и начинается исполнение программы. Т. е. правило (H13) непротиворечиво для частичной корректности.

■

6. ЗАКЛЮЧЕНИЕ

Описанный в настоящей работе язык C-light-kernel является промежуточным языком в двухуровневой схеме верификации C-light программ и непосредственным входом для блока генерации условий корректности. Являясь ядром языка C-light и, следовательно, подмножеством языка ANSI C, он позволяет существенно упростить аксиоматичекую семантику входных программ, сохраняя при этом большую часть

выразительной мощи языка C. Кроме того программы на языке Паскаль естественным образом транслируются в него, что позволяет проводить верификацию программ на этом языке, используя опыт системы СПЕКТР [4].

Важным достоинством языка C-light-kernel является то, что указатели языка C-light сохраняются в нем практически полностью. Для указателей была разработана проблемная область в виде аксиом, определяющих свойства ссылочных классов.

Помимо создания операционной и аксиоматической семантик, важным результатом является доказательство свойства непротиворечивости аксиоматической семантики HSC относительно операционной. На основе непротиворечивой и небольшой по объему системы HSC был создан прототип генератора условий корректности, позволяющий автоматически порождать утверждения о свойствах программ — условия корректности.

Заметим, что эта выбранная двухуровневая схема предусматривает также формальное обоснование корректности трансляции языка C-light в язык C-light-kernel. Для этого необходимо полностью формально определить правила перевода и индуктивно доказать, что в процессе трансляции семантика программы сохраняется. Это доказательство станет основной темой третьей части работы, посвященной верификации программ на языке C.

СПИСОК ЛИТЕРАТУРЫ

1. **Керниган Б., Ритчи Д.** Язык программирования Си. — М.: Финансы и статистика, 1985.
2. **Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.** На пути к верификации C программ. Часть 1. Язык C-light. — Новосибирск, 2001. — (Препр. / РАН. Сиб. Отд-ние. ИСИ; N 84).
3. **Непомнящий В.А., Рякин О.М.** Прикладные методы верификации программ. — М.: Радио и связь, 1988.
4. **Непомнящий В.А., Сулимов А.А.** Проблемно-ориентированные базы знаний и их применение в системе верификации программ СПЕКТР // Известия РАН. Сер. "Теория и системы управления". — 1997. — N 2. — С. 169–175.
5. **Apt K.R., Olderog E.R.** Verification of sequential and concurrent programs. — Berlin a.o.: Springer Verlag, 1991.
6. **Black P.E., Windley Ph.J.** Inference rules for programming languages with side effects in expressions // Proc. 9th Intern. Conf. on Theorem Proving in HOL. — Berlin a.o.: Springer Verlag, 1996. — P. 56–60. — (Lect. Notes Comput. Sci.; Vol. 1125).
7. **Elgaard J., Moller A., Schwartzbach M.I.** Compile-time debugging of C

programs working on trees // Proc. Europ. Symp. on Programming (ESOP2000). — Berlin a.o.: Springer Verlag, 2000. — P. 119–134. — (Lect. Notes Comput. Sci.; Vol. 1782).
8. **Fradet P., Gaugne R., Le Metayer D.** Static detection of pointer errors: an axiomatisation and a checking algorithm // Proc. Europ. Symp. on Programming (ESOP96). — Berlin a.o.: Springer Verlag, 1996. — P. 125–140. — (Lect. Notes Comput. Sci.; Vol. 1058).
9. **Gurevich Y., Huggins J.K.** The semantics of the C programming language // Proc. of the Intern. Conf. on Computer Science Logic. — Berlin a.o.: Springer Verlag, 1993. — P. 274–309. — (Lect. Notes Comput. Sci.; Vol. 702).
10. **Huggins J.K., Shen W.** The static and dynamic semantics of C (extended abstract) // Local Proc. Int. Workshop on Abstract State Machines. — Zurich, 2000. — P. 272–284. — (ETH TIK-Rep.; N 87).
11. **Luckham D., Suzuki N.** Verification of array, record and pointer operations in Pascal // ACM Trans. on Programming Languages and Systems, 1979. — Vol. 1, N 2. — P. 226–244.
12. **McCarthy J.** Towards a mathematical science of computation // Proc. IFIP Congress 62. — Amsterdam: North-Holland Publ. Co., 1962. — P. 21–28.
13. **Norrish M.** Deterministic expressions in C // Proc. Europ. Symp. on Programming (ESOP99). — Berlin a.o.: Springer Verlag, 1999. — P. 147–161. — (Lect. Notes Comput. Sci.; Vol. 1576).
14. **Norrish M.** C formalised in HOL // PhD thes., Computer Lab., Univ of Cambridge, 1998.
15. **Plotkin G.D.** A structure approach to operational semantics: Technical Rep. FN-19, Aarhus University, DAIMI, 1981.
16. **Suzuki N.** Analysis of pointer Rotation // Seventh Ann. ACM Symp. on POPL, 1980.
17. **Wilson R.P., Lam M.S.** Efficient Context-Sensitive Pointer Analysis for C Programs // ACM SIGPLAN Notices, 1995. — Vol. 30. — N 6. — P. 1–12.

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

НА ПУТИ К ВЕРИФИКАЦИИ С-ПРОГРАММ. ЧАСТЬ 2. ЯЗЫК C-LIGHT-KERNEL И ЕГО АКСИОМАТИЧЕСКАЯ СЕМАНТИКА

Препринт
87

Рукопись поступила в редакцию 07.05.2001
Рецензент Ф. А. Мурзин
Редактор З. В. Скок

Подписано в печать 01.06.2001
Формат бумаги 60×84 1/16 Объем 3,3 уч.-изд.л., 3,6 п.л.
Тираж 50 экз.

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