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 1. ЯЗЫК C-LIGHT

Препринт
84

Новосибирск 2001

Описано ориентированное на верификацию представительное подмножество языка C — язык C-light. Этот язык допускает детерминированные выражения, ограниченное использование операторов *switch* и *goto*, а вместо библиотечных функций для работы с динамической памятью включает операции *new* и *delete* языка C++. Для языка C-light представлена структурная операционная семантика в стиле Плоткина. Предполагается использование языка C-light в качестве входного языка системы верификации программ.

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

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

TOWARDS C PROGRAMS VERIFICATION. PART 1. LANGUAGE C-LIGHT

Preprint
84

Novosibirsk 2001

A language C-light which is a representative subset of the language C is proposed. This language includes deterministic expressions, statements switch and goto with some restrictions as well as the functions new and delete for processing the dynamic memory. It imposes some restrictions on the use of types of the language C. Complete structural operational semantics in Plotkin style is presented for C-light. Using C-light as an input language of a program verification system is suggested.

1. ВВЕДЕНИЕ

Верификация программ — одно из актуальных направлений современного программирования. Обозримая формальная семантика является необходимой предпосылкой того, что язык программирования удобен для верификации [2, 4]. В качестве таковых выступают языки, которые подобно Паскалю не находят широкого применения при разработке больших надежных программных систем. Для таких целей широко используется язык C++.

Язык C++ не удобен для верификации, так как для него не существует формальной семантики. Поэтому актуальна проблема выделения удобного для верификации представительного подмножества C++. Такую проблему естественно исследовать по отношению к языку C [1]. Хотя формальной семантики для полного языка C, соответствующего стандарту ANSI, также не существует, она была предложена для довольно представительного подмножества C, в котором накладывались, в частности, естественные ограничения на вычисление выражений [8, 9]. Заметим, что предложенная в этих работах формальная семантика не ориентирована на верификацию.

Трудности верификации C-программ обсуждались в [5–7], где рассматривались только отдельные конструкции языка C. В [5] изучались выражения и функции с побочными эффектами, оператор присваивания и while-циклы, для которых была предложена аксиоматическая семантика т. е. правила вывода условий корректности в стиле Хоара. Верификация C-программ над указателями рассматривались в [6], где был сделан упор на алгоритмические методы верификации безцикловых программ. Метод анализа C-программ над указателями предложен в [7].

Представительное подмножество языка C, для которого разработана структурная операционная семантика, рассматривалось в [10, 11]. Заметим, что это подмножество не включает библиотечных функций, операторов switch и goto, а также некоторых типов, как, например, объединения. Достоинство работ [10, 11] состоит в исследовании семантики выражений, которые могут быть недетерминированными. Кроме того, операционная семантика ориентирована на верификацию благодаря вложению этой семантики во входной язык системы доказательства теорем HOL, а также обоснованию нескольких правил аксиоматической семантики. Однако отсутствие библиотечных функций не позволяет про-

водить верификацию С-программ, которые работают с динамической памятью.

Цель настоящей работы — предложить ориентированный на верификацию язык C-light, который допускает ограниченное использование операторов switch и goto, а вместо библиотечных функций включает функции new и delete языка C++ для работы с динамической памятью.

В данной работе в разд. 2 излагаются основные принципы, а также ограничения на язык С, принятые в нашем подмножестве C-light. Разд. 3 посвящен вводному обзору языка C-light, где кратко описаны как язык программирования, так и язык аннотаций. Описанию статической семантики языка C-light посвящен разд. 4. Динамическая семантика этого языка представлена в разд. 5. В заключение обсуждаются перспективы развития метода верификации программ на языке C-light. Приложение содержит формальный синтаксис языка C-light.

Работа частично поддержана грантом РФФИ 00-01-00909.

2. ПРИНЦИПЫ И ОГРАНИЧЕНИЯ

Исходный код программы на С не однозначно транслируется в исполнимый код в отличие от, скажем, языка Java. Различные реализации ANSI C могут порождать из одного и того же исходного кода различные, хотя и похожие варианты объектного кода, который в свою очередь не абсолютно однозначно транслируется в исполнимый код. Для текста программы, написанного с некоторыми предосторожностями, эти различия оказываются достаточно малыми и не влияют на возможность практического использования исполнимой программы, но для целей верификации неоднозначность должна быть существенно уменьшена.

Рассмотрим возможные источники неоднозначности результата прогона исходного текста, транслированного в исполнимый код.

1. Текст на языке ANSI C не определяет однозначно схему программы. Например, выражение `f()+g()` может быть вычислено двумя способами: "вызвать `f()`, вызвать `g()`, сложить результаты", либо "вызвать `g()`, вызвать `f()`, сложить результаты". Если побочный эффект при вызове одной из этих функций влияет на работу другой, то результат прогона программы зависит от того, какой вариант изберет транслятор. Это затрудняет верификацию аннотированной программы, так как для каждого подобного выражения требуется отдельно верифицировать каждый вариант трансляции.

2. Даже при однозначно определенной схеме программы результат программы может зависеть от таких параметров целевой машины, как разрядность целочисленной и адресной арифметики целевой машины, разрядность и точность вычислений с вещественными значениями, разбиение памяти на сегменты, направление роста стека, схема выравнивания данных по границам слов, «байтордер» и пр. К счастью, если данные параметры влияют на результат прогона программы, то это влияние хорошо предсказуемо и аннотации могут быть записаны как выражения от этих параметров.

3. Даже при однозначно определенных схеме программы и параметрах целевой машины результат программы может зависеть от начального состояния памяти и другой аппаратуры и от размещения программы в памяти. Если такая зависимость имеет место, то либо в программе имеется логическая ошибка (например, обращение к неинициализированной памяти, запись данных поверх исполнимого кода, выход за границы массива), либо аннотация должна также зависеть от начального состояния машины (например, если программа должна печатать дамп памяти).

4. Разные трансляторы могут порождать код разной эффективности, что может привести к различным временам работы программы. C-light не разрабатывался для верификации систем реального времени, поэтому в общем случае этот вопрос остается открытым; но если ни программа ни ее аннотация не привязаны к астрономическому времени, эффективность кода не может влиять на истинность условий корректности.

Таким образом, важнейшим требованием к языку оказывается требование единственности схемы программы. В C-light порядок построения схемы программы однозначно описан для каждого случая возможной неоднозначности, например, полностью фиксирован порядок вычисления выражений.

Однако даже устранение недетерминизма в выражениях не снимает главного недостатка стандартного определения ANSI C. Дело в том, что стандарт большей частью написан на обычном английском языке, а значит, не является полностью формальным. С другой стороны, верификация программ является полностью формальным процессом, что гарантирует корректность результатов, поэтому на основе стандарта необходимо создать формальную, логически непротиворечивую семантику.

Помимо детерминизации был сделан ряд ограничений, касающийся главным образом низкоуровневых возможностей языка C. Некоторые конструкции мы не ограничиваем, но упрощаем их семантику. Перечислим основные ограничения в текущей версии системы.

- **Типы**.
  - Не поддерживаются объединения и битовые поля в структурах.
  - Элементы перечислимых типов являются объектами типа signed int.
  - По умолчанию типы int и wchar_t имеют знак, поведение типа char определяется флагами компиляции.
- **Декларации**.
  - Запрещены «неокончательные определения» (tentative definitions).
  - Запрещены почти все модификаторы типа и спецификаторы, кроме спецификаторов наличия знака и спецификаторов размера для скалярных типов.
- **Указатели**.
  - Используется плоская модель памяти.
  - Все указатели 32-битные.
  - Запрещены указатели на функции.
  - Приведение типов для указателей только через тип void.
  - Для работы с динамической памятью используются операции языка C++.
- **Операции**. Игнорируются все побитовые операции.
- **Выражения**. В выражении присваивания на верхнем уровне запрещена операция последовательного вычисления.
- **Функции**.
  - Не поддерживаются функции стандартной библиотеки.
  - Запрещены функции с переменным числом аргументов.
  - Запрещены абстрактные декларации аргументов.
  - Запрещены аргументы по умолчанию.
- **Операторы**.
  - Передача управления по goto может происходить либо в пределах блока, либо из вложенного блока в охватывающий, но не наоборот.
  - Метки в операторе switch должны находиться на одном уровне вложенности.

Таким образом, основные принципы, использованные при определении языка C-light, — это устранение неоднозначности и упрощение семантики некоторых типов и операций. Тем самым мы избавляемся от излишней громоздкости семантики, которая может стать существенным препятствием для верификации больших программ [11]. Подробнее некоторые ограничения рассмотрены в следующих пунктах.

3. ВВОДНЫЙ ОБЗОР ЯЗЫКА C-LIGHT

Язык аннотированных программ C-light является подмножеством «чистого» C, расширенным некоторыми лексическими правилами из C++ и аннотациями. Любой текст, соответствующий одновременно синтаксису C-light и синтаксису ANSI C, будет иметь одинаковый смысл в обоих этих языках. При дальнейшем описании отличия от ANSI C будут выделяться особо.

Так же как и обычный C, язык C-light 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 нет неявно вводимой «виртуальной машины», умеющей манипулировать такими объектами высокого уровня, как, например, символьные строки. Знание этого факта позволяет описывать язык, сопоставляя каждой конструкции C-light некоторое понятие фон-неймановской вычислительной машины с линейной памятью.

Следует сразу отметить, что большинство деталей языка C++, допустимых в формальном определении синтаксиса, не поддерживаются в полном объеме текущей реализацией семантики. В основном они были введены для того, чтобы в будущем облегчить переход к языку C++.

В п. 3.1 приведено неформальное описание языка описания программ, т. е. той части языка C-light, на которой пишется код программ. Эта часть является пояснением к правилам формального синтаксиса, рассмотренного в приложении. В п. 3.2 излагается язык описания аннотаций, т. е. язык спецификации программ в системе СПЕКТР.

3.1. Язык программ

Исходная программа. Весь исходный текст программы есть последовательность внешних деклараций. Как обычно, на внешнем уровне можно объявить тип (typedef-декларации), функцию (прототип или определение), структуру, перечисление (в отличие от раннего стандарта ANSI

C) и данные. Кроме обычных конструкций C на верхнем уровне могут присутствовать и некоторые типы аннотаций.

Синтаксический анализ исходного текста программы выполняется в несколько проходов. Первый проход пропускает тела функций. В последующих проходах, напротив, анализируются только тела. Более точно, сначала в списке всех лексем исходного текста выделяется участок, соответствующий телу функции, вместе с обрамляющими его фигурными скобками; затем к нему применяется правило для тела.

В базовой реализации C-light препроцессор не предусмотрен. По своему усмотрению пользователь может использовать препроцессоры C, C++ или любой другой или не использовать вовсе, что может быть предпочтительным. Несмотря на отсутствие препроцессора, комментарии использовать можно.

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. Однако верификация стандартной библиотеки возможна только при наличии исходных файлов, так как вызов функции в выражении требует перехода к телу функции, поэтому, например, функции работы с динамической памятью были заменены на операции языка C++.

Аннотации. В отличие от C в C-light 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 без лексических ошибок и в нем соблюден «баланс скобок». Используются аннотации для записи спецификаций программ.

Разные системы верификации программ могут использовать разные грамматики для придания смысла аннотациям. Базовая реализация C-light делает с аннотациями две операции:

1. При чтении программы каждая аннотация разбивается на лексемы.

2. При преобразовании выражений программы каждое вхождение выражения C-light в некоторую аннотацию заменяется вхождением преобразованного выражение, как если бы это выражение было частью исполнимого кода, например: замена переменной `pi` на константу 3.1415 во всех выражениях программы приведет к тому, что все вхождения `pi` во все аннотации будут также заменены на 3.1415.

Аннотации делятся на обычные и управляющие. Обычная аннотация есть тело аннотации, обрамленное специальными скобками:

```
/% true %/      // Аннотация "Всегда истинно"
```

Управляющая аннотация позволяет определить способ обработки тела функции. Тройки Хоара для inline-функций будут доказываться обязательно, тройки extern-функций — 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.

```
/% inline int main(int argc, const char *argv[]) %/
                // Для головной функции обычно
                // требуется управляющая аннотация
```

Тело аннотации есть почти произвольная последовательность лексем. Ограничений всего три: должен соблюдаться баланс скобок, в теле не может содержаться %%, внутри скобок (% ... %) может содержаться только правильное выражение C-light.

```
foreach i in (%1%) .. (%N+1%) A[i] < (%N%)
                // Допустимое тело аннотации
```

Имена. Поскольку язык C-light содержит некоторые детали языка C++, то имена в нем несколько отличаются от имен языка C.

В «полном» C++ имя класса есть либо полное имя конкретной реализации шаблона, либо идентификатор. В C-light это просто идентификатор.

В C имя объекта не может быть «квалифицировано» операцией ::, в C++ есть дополнительно понятие «имя деструктора» (с тильдой) и имена вида "operator ...". Текущая версия системы C-light поддерживает "четвероточие", но не поддерживает "operator ...".

Примеры:

```
struct not_yet_defined // Квалифицированное имя класса
unsigned long          // Стандартный тип
ofstream::flush        // Ранее определенное имя
FILE *                 // Указатель на ранее созданный typedef
const math::complex    // Константный объект
```

Декларации. Рассмотрим некоторые примеры деклараций в языке C-light и сформулируем ряд ограничений.

Декларации устанавливают соответствие между идентификаторами и объектами в памяти. В C-light, 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 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.

ANSI C вводит понятие «*неокончательного определения*» (**tentative definition**). ANSI C++ отменяет это понятие, считая дополнительное «окончательное определение» объекта синтаксической ошибкой. В языке C-light понятие неокончательного определения отсутствует.

```
int main(int argc, const char *argv[]);
                // Декларация функции
typedef struct cplx { double re, im; } COMPLEX;
                // Одновременная декларация
                // структуры и typedef
```

```
int *a, b=0, c[3];
                // Декларация трех переменных:
                // указателя на целое, целого и массива целых
typedef signed long unicode_symbol;
                // Присвоение дополнительного
                // имени базовому типу
enum digits { zero, one, two, three, four=4, five, six,
    seven=six+1, eight, nine, ten }
                // Перечисление для десятичных цифр
```

Объект может быть полностью или частично инициализирован.

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

Составной объект, такой как структура или массив, может быть инициализирован только константными выражениями и только если он является статическим. Объединение не может быть инициализировано.

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

Как и в C, одна декларация может содержать целый список декларируемых объектов. Фрагмент, относящийся к конкретному объекту, содержит имя объекта и его «персональные» модификаторы типа. В начале декларации перед первым таким фрагментом уже был указан «основной» тип декларации; последовательно применяя «персональные» модификаторы к этому «основному» типу, компилятор определяет тип объекта, декларируемого в данном фрагменте. Рассмотрим следующую декларацию:

```
char c1, c2 ='a', c3[] = "abc", *c4[2], c5;
```

1. Первый фрагмент описывает переменную без модификаторов: `c1`.
2. Второй фрагмент описывает такую же переменную (инициализатор в него не входит): `c2`.

3. Третий фрагмент — массив, размер которого будет определен при инициализации: `c3[]`.
4. Четвертый фрагмент — массив из двух указателей: `*c4[2]`.
5. Последний фрагмент — скаляр, как и первые два: `c5`.

Различия между class и struct влияют на синтаксическую корректность программы на C++, но никак не влияют на генерируемый код. В C-light class и struct не различаются вовсе, а union отличается только способом вычисления смещений полей и, соответственно, способом вычисления sizeof. Текущая версия системы не может строить условия корректности для кода, в котором используются типы, созданные с помощью union.

Список членов структуры может содержать спецификаторы доступа (они будут игнорироваться) и битовые поля (текущая версия системы не может создать условия корректности для кода, в котором используются структуры битовыми полями).

Несмотря на отсутствие в C-light контроля за правами доступа, использование в декларации спецификатора friend существенно меняет семантику. Если имя объекта, декларируемого со спецификатором friend не определено локально внутри декларации структуры, то он будет создан не внутри декларации структуры (как, например, еще один член), а вне ее.

Модификатор "функция, возвращающая значение типа ..." не только описывает тип, но и может назначить имена некоторым объектам — аргументам функции. Список аргументов может заканчиваться троеточием, однако в семантике такие функции не поддерживаются. Пустой список аргументов не разрешен — требуется void.

Декларация отдельного аргумента функции может быть либо «обычной», либо абстрактной декларацией. Кроме того, может быть указано значение по умолчанию, но текущая версия системы не сможет создать условия корректности для вызовов функций с опущенными необязательными аргументами. В следующем примере декларация первого аргумента абстрактная, второго — обычная:

```
int main (int, const char argv[]);
```

Декларация функции может содержать тело. В этом случае синтаксический анализ отдельных фрагментов этой конструкции будет выполняться в последующие проходы.

Спецификация декларации перечисляет свойства объекта, которые нужны в основном для оптимизации кода. Все они игнорируются теку-

щей версией системы, и большинство будет игнорироваться и в будущем, так как операционная среда C-light существенно проще реальной операционной среды C. Рассмотрим подробнее все игнорируемые спецификаторы и модификаторы:

• Спецификатор размещения объекта в объектном коде: auto, register, static, extern.

• Спецификатор inline игнорируется, так как результат его действия зависит от компилятора, а virtual не может иметь эффекта, так как наследование не поддерживается.

• Спецификатор наличия Run Time Type Information __rtti; попытка включить поддержку RTTI приводит к фатальной ошибке.

• Спецификаторы модели памяти класса: near, far, huge, __exept, __import. Это связано с тем, что в C-light имеется только линейная модель памяти, и каждый из этих спецификаторов либо запрещен, либо не имеет эффекта.

• Модификаторы const и volatile. Модификатор const влияет на семантику только при обработке обрезки аргумента (*slicing*), что не существенно, пока не поддерживается наследование классов. Модификатор volatile не меняет семантику синхронного кода, в то время как асинхронность в C-light невозможна вообще.

• Спецификаторы формата стека для вызова функции, кроме cdecl и pascal, так как прерывания отсутствуют, а спецификаторы размера указателя не имеют смысла для линейной памяти C-light.

• Спецификаторы доступа private, protected и public влияют на синтаксическую корректность программы на C++, но никак не влияют на генерируемый код. В C-light они просто игнорируются, что не меняет семантику.

Выражения. Главным отличием языка C-light от C являются наличие детерминизма вычисления и запрет низкоуровневых операций. Подробнее синтаксис выражений рассматривается при определении статической семантики и в приложении.

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

```
tmp = a, a = b, b = tmp
                // Обмен значений двух переменных,
                // записанный как одно выражение
```

Выражение присваивания может либо содержать операции присваивания, либо быть простым rvalue. Главное, что в нем на верхнем уровне нет операции «запятая», поэтому, в отличие от выражения общего вида, выражение присваивания может встречаться в различных списках выражений:

```
a = b = c = 0   // Цепочка присваиваний тоже есть
                // выражение присваивания
```

В C-light предусмотрены все операции присваивания C, но условия корректности не могут быть сгенерированы для выражений с побитовыми операциями.

Полностью запрещенными являются операции $.*$ и $->*$.

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

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

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

Пустой оператор тоже является оператором вычисления выражения.

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

Мы считаем, что в условном операторе if всегда есть ветвь else, возможно пустая. На метки в операторе switch налагается ограничение. Все метки должны находиться на одном уровне вложенности, т. е. запрещен следующий вариант:

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

Причина такого ограничения подробнее рассмотрена при описании правил операционной семантики для этого оператора.

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.

3.2. Язык аннотаций

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

3.2.1. Типы и алфавит

Допустимыми типами выражений языка аннотаций являются следующие:

- **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 $\mathbf{Ref}(T)$, а ptr — указатель типа $T*$,
присваивание:	$\langle$X, ptr, value$\rangle$,
расширение:	$\mathrm{X} \cup \{\mathrm{address}_1, \mathrm{address}_2\}$, где $\mathrm{address}_1 \leq \mathrm{address}_2$,
сужение:	$\mathrm{X} \setminus \{\mathrm{address}\}$.

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 $\cup$.

3.2.2. Переменные и константы

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 обозначим $\mathbf{P}\#T$. 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 называются переменными, поскольку они могут находиться в левой части операции

присваивания и быть аргументами в подстановке. По терминологии [7] это *subscripted variables*.

4. СТАТИЧЕСКАЯ СЕМАНТИКА

Рассмотрим подробнее синтаксис выражений языка C-light, одновременно определяя правила типизации. Тем самым мы определим статическую семантику языка C-light. В статической семантике не рассматриваются операторы, поскольку, хотя у них и есть значения — преобразователи состояний, эти значения только неявно присутствуют в абстрактной машине. Статическая семантика выполняет две важные функции. Во-первых, в динамической семантике, т. е. в семантике исполнения программы, необходимо знать типы выражений. Чтобы не усложнять динамическую семантику, будем считать, что ей на вход подается программа, в которой типы уже определены статической семантикой. Во-вторых, статическая семантика может служить специальным «фильтром», который отлавливает ошибки в системе типов. Можно считать, что на вход динамической семантике подается программа, корректная с точки зрения типов.

Как уже отмечалось, любое выражение языка C-light может быть частью аннотации. Выражения имеют типы и любое выражение непустого типа имеет значение, которое может быть и неопределенным — ω.

Как и в C, значением выражения может быть ссылка на объект в памяти — так называемое L-значение (lvalue). Для спецификации того, что выражение является L-значением, используется запись $\mathrm{LV}[T]$.

Для определения типов выражений нам понадобятся три конечных отображения:

1) Γ — отображение из имен переменных в типы, исключая пустой тип;

2) Φ — отображение из имен функций в типы;

3) Σ — отображение, сопоставляющее тегу произвольной структуры последовательность пар имен и типов полей этой структуры.

В дальнейшем эти отображения войдут также в определение состояния.

Выражения языка спецификаций определяются по индукции. Для изображения чисел используется символ n, для символов — c, для переменных, функций и членов структур — символ id. Выражение отделя-

ется от его типа двоеточием. Базой являются следующие аксиомы для констант и переменных:

n : signed int
nL : long
nU : unsigned int
nS : short int

$n_1.n_2[\mathsf{E}\, n_3]$: double
$n_1.n_2[\mathsf{E}\, n_3]$F : float
$n_1.n_2[\mathsf{E}\, n_3]$L : long double

'c' : char
L'c_1c_2' : wchar_t
"$c_1 \ldots c_n$" : char$[n+1]$
L"$c_1 \ldots c_n$" : wchar_t$[n/2+1]$

NULL : void$*$
$0 : T*$

var id : LV$[\Gamma(\mathsf{id})]$
function id : $\Phi(\mathsf{id})$

Следующие правила относятся к операциям адресации и косвенной адресации. Отметим сразу, что нет отдельного правила для индексных выражений, так как выражение `a[i]` есть просто удобная запись для $*(\mathtt{a}+\mathtt{i})$.

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

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

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

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

сразу, что запись вида `ptr->f` обрабатывается как `(*ptr).f`.

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

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

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

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

Для корректной обработки вызовов функций и присваиваний необходимо ввести отношение приводимости по умолчанию (**implicit coercibility**). В противном случае в правилах для операции присваивания и вызова функции пришлось бы явно записывать требования точного согласования типов аргументов. Это отношение, обозначаемое как IC, будет отношением эквивалентности. Во втором правиле одним из типов может быть тип bool.

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

Тогда правило для вызова функции выглядит как

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

Что касается операций языка C-light, то правила для них могут быть достаточно громоздкими, потому что над аргументами происходят стандартные преобразования типов. Поэтому информация об окончательном типе выражения определяется специальными предикатами. Для обозначения унарных операций используем символ $\square$, для бинарных операций — символ $\odot$. Заметим, что операции постфиксного инкремента/декремента рассмотрены отдельно.

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

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

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

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

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

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

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

Наконец, необходимо задать правила для операций над ссылочными классами:

$$\frac{e_1 : \mathbf{Ref}(T) \qquad e_2 : T* \qquad T \neq \mathsf{void}}{e_1[e_2] : T}$$

$$\frac{e_1 : \mathbf{Ref}(T) \qquad e_2 : T* \qquad e_3 : T}{\langle e_1, e_2, e_3 \rangle : T}$$

$$\frac{e_1 : \mathbf{Ref}(T) \qquad e_2 : T*}{e_1 \cup \{e_2, e_2 + n\} : \mathbf{Ref}(T)} \qquad \frac{e_1 : \mathbf{Ref}(T) \qquad e_2 : T*}{e_1 \setminus \{e_2\} : \mathbf{Ref}(T)}$$

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

5. ДИНАМИЧЕСКАЯ СЕМАНТИКА ЯЗЫКА C-LIGHT

5.1. Предварительные определения

Метод верификации состоит в формальном доказательстве утверждений о свойствах программ. Для того чтобы это доказательство стало возможным, необходимо придать смысл конструкциям языка программирования. При этом описание смысла должно быть полностью формальным и логически непротиворечивым. Существует три основных подхода: *операционный*, *денотационный* и *аксиоматический*. При первом определяется абстрактная вычислительная машина, которая интерпретирует синтаксис языка. Фактически при этом моделируется реальное исполнение программы. При денотационном подходе в качестве модели языка определяется подходящее математическое пространство, и синтаксис отображается в это пространство. Аксиоматический подход — самый абстрактный — определяет язык как исчисление троек Хоара.

В нашем случае выбран первый подход, т. е. формальным определением языка C-light является структурная операционная семантика предложенная в работе [12]. При создании семантики также использовались работы [4, 11].

5.1.1. Состояния абстрактной машины

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 T фиксируем множество значений, называемое *носителем* типа T и обозначаемое D_T. Поскольку мы не ориентируем семантику 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 T;
- $D_{T[\textbf{n}]} = D_T^n$ (декартова степень n) для любого непустого и нефункционального типа T;
- $D_{\textbf{struct}(T_1\ v_1;\ \ldots;\ T_n\ v_n)} = D_{T_1} \times \ldots \times D_{T_n}$;
- $D_{\textbf{Ref}(T)} = D_{\textbf{unsigned int}} \to D_T$; т. е. множество всех функций из указателей в значения типа 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 T фиксируем значение в носителе D_T и говорим, что константа *обозначает* это значение. Считаем, что каждая константа базового типа (а также типов «массив» и «структура») обозначает саму себя. В свою очередь константы типов «функций» обозначают соответствующие функции.

В отличие от констант значения переменных не фиксированы и определяются через состояния абстрактной вычислительной машины. *Состояние* — это в простейшем случае отображение, которое присваивает любой переменной типа T значение в области D_T. Используем слово *States* для обозначения множества всех состояний. 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 модели-

ровать какую-либо конкретную архитектуру и не изучаем раскладку данных в памяти.

Семантика $\mathcal{I}\|s\|$ выражения s типа 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 T', 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 C-light, например, бинарную операцию "+"можно считать константой типа **int**$\times$ **int** $\to$ **int**.

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

Замечание. Приведение типов — один из сложных моментов в семантике выражений. Чтобы корректно интерпретировать эту операцию, необходимо явно привязываться к конкретной архитектуре (см., например, [18]), что усложнит семантику. Кроме того, эта операция определена не для любых типов, а указывать в семантике условия ее применимости громоздко. Поэтому мы рассматриваем только безопасные приведения скалярных типов и указателей (через пустой тип) и для них задаем семейство функций $\gamma_{T',T}$.

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 понятие *истинности* 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, а также используем понятие смысла утверждения, определенного как

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

Мы говорим, что утверждение p *истинно* или *выполняется*, если $\|p\| = States$.

5.1.2. Конфигурации абстрактной машины

Операционная семантика языка программирования представляется в виде множеств пар конфигураций абстрактной вычислительной машины. Конфигурация — это пара $\langle \mathrm{P}, s \rangle$ — программа P и состояние s. Пустая программа обозначается символом E, и записи E; A и A; E являются эквивалентными обозначениями для программы A.

Как уже отмечалось ранее, в простейшем случае состояние — это отображение из множества идентификаторов переменных во множество их потенциальных значений. Такого представления достаточно для простых модельных языков [4], но для языка C-light необходимо добавить новые компоненты.

Определение. Состояние абстрактной вычислительной машины языка C-light — это семерка {M, Γ, Σ, Φ, TD, П, GLF}, 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 (уже упоминалось в определении статической семантики);

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;

6) П — очередь отложенных побочных эффектов, которые накапливаются при вычислении выражений. Очередь очищается при достижении контрольной точки. Отметим, что если бы мы не ввели детерминизм вычисления выражений, то очередь превратилась бы в мультимножество;

7) GLF — флаг, определяющий вложенность, необходим для корректной инициализации переменных, поскольку спецификаторы классов памяти игнорируются. В начале работы программы, т. е. на глобальном уровне, флаг равен нулю, при входе в любой блок увеличивается на единицу, при выходе соответственно уменьшается.

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: $\sigma(X(a \leftarrow e), Y(b \leftarrow i))$. Такая запись имеет смысл для компонент, являющихся отображениями. Для флага GLF, например, можно использовать просто символ равенства.

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

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

2) отношение для вычисления объявлений: $\rightarrow_v$,

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

Будем также считать, что значения сопровождаются их типами, т. е. пара (value, type) тоже является значением. Типы определяются статической семантикой, рассмотренной ранее.

Итак, опишем все аксиомы и правила операционной семантики языка C-light. Аксиомы будут изображаться в виде пар конфигураций, связанных одним из отношений перехода, и запись $\langle \mathrm{A}, \sigma \rangle \rightarrow \langle \mathrm{B}, \tau \rangle$ означает, что один шаг исполнения фрагмента A исходной программы, 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. Посылки в правилах семантики отделяются от заключений чертой.

5.2. Выражения

5.2.1. Контекст вычисления выражений

Очевидно, что разнообразие операций языка C-light может стать причиной громоздкости операционной семантики. Однако очевидно, что большинство операций можно обрабатывать единым образом. Для этого вводится понятие контекста вычислений [11]. Неформально контекст вычисления — это фрагмент синтаксиса с «пробелом в нем». На место пробела можно подставлять выражения. Определим контекст:

$$\begin{aligned}\mathcal{E}[-] ::= \quad & -\odot e \mid e\odot - \mid \square - \mid -\,\&\&e \mid -\,||\,e \mid -, e \mid \\ & -\,\texttt{++} \mid -\,\texttt{--} \mid -.\mathsf{id} \mid -\odot= e \mid -= e \mid (\tau) - \mid \\ & -?e_1 : e_2 \mid -(e_1, ..., e_n) \mid e(e_1, ..., -, ..., e_n)\end{aligned}$$

Тогда следующее правило обрабатывает вычисление подтермов в выражении:

$$\frac{\langle e_0, \sigma_0\rangle \to_{\mathrm{e}} \langle e, \sigma\rangle}{\langle \mathcal{E}[e_0], \sigma_0\rangle \to_{\mathrm{e}} \langle \mathcal{E}[e], \sigma\rangle}. \tag{1}$$

Неопределенное значение может протягиваться из контекста:

$$\langle \mathcal{E}[\omega], \sigma_0\rangle \to_{\mathrm{e}} \langle \Omega, \sigma\rangle. \tag{2}$$

Контекст не распространяется дальше правой части (составного) оператора присваивания, поэтому для неопределенного значения введем еще одно правило (для составного присваивания правило выглядит так же):

$$\langle e_1 = \omega, \sigma\rangle \to_e \langle \Omega, \sigma\rangle. \tag{3}$$

5.2.2. Значения из памяти

Константы:

$$\frac{\Gamma_\sigma(\mathrm{const}) = \tau}{\langle \mathrm{const}, \sigma\rangle \to_e \langle (\mathrm{M}_\sigma(\mathrm{const}), \tau), \sigma\rangle}. \tag{4}$$

Переменные и L-значения:

$$\frac{\Gamma_\sigma(\mathsf{id}) = \tau - \text{не массив}}{\langle \mathsf{id}, \sigma\rangle \to_e \langle (\mathrm{M}_\sigma(\mathsf{id}), \tau), \sigma\rangle}. \tag{5}$$

Массивы:

$$\frac{\Gamma_\sigma(\mathsf{arr_id}) = \tau[n]}{\langle \mathsf{arr\_id}, \sigma\rangle \to_e \langle(\mathsf{arr\_id}, \tau*), \sigma\rangle}. \tag{6}$$

Структуры:

$$\frac{(\mathsf{id}, \tau) \in \Sigma_\sigma(s)}{\langle \mathsf{struct}\ s.\mathsf{id}, \sigma\rangle \to_e \langle(\mathrm{M}_\sigma(\mathsf{id}), \tau), \sigma\rangle}. \tag{7}$$

5.2.3. Операции, выдающие значения

Для того чтобы не дублировать правила для стандартных унарных и бинарных операций, не являющихся контрольными точками, вычисление значений производится специальными функциями UnOpSem и BinOpSem, которые реализованы на языке ML в полном соответствии со стандартом ANSI и здесь не приводятся.

$$\frac{\odot \in \{\text{стандартные бинарные операции}\} \setminus \{\&\&, ||\text{ и операция ,}\}}{\langle(v_1, \tau_1) \odot (v_2, \tau_2), \sigma\rangle \to_e \langle \mathrm{BinOpSem}(\sigma, \odot, (v_1, \tau_1), (v_2, \tau_2)), \sigma\rangle}. \tag{8}$$

$$\frac{\square \in \{\tilde{}, !, -\}}{\langle \square(v, \tau), \sigma\rangle \to_e \langle \mathrm{UnOpSem}(\square, (v, \tau)), \sigma\rangle}. \tag{9}$$

$$\frac{\gamma_{\tau_0,\tau}(v_0) = v}{\langle(\tau)(v_0, \tau_0), \sigma\rangle \to_e \langle(v, \tau)), \sigma\rangle}. \tag{10}$$

$$\frac{\Pi_\sigma = \emptyset}{\langle(v, \tau), e, \sigma\rangle \to_e \langle e, \sigma\rangle}. \tag{11}$$

Логические операции И и ИЛИ, а также условная операция выделены в отдельную группу. Как известно, выражения, содержащие эти операции, могут вычисляться не полностью[1], поэтому такие операции

[1] Способ вычисления логических операций определяется настройками компилятора.

являются контрольными точками. Для вычисления этих операций вводится промежуточная форма **OrAnd**.

$$\frac{\tau - \text{скалярный тип} \qquad \mathrm{M}_\sigma(v_0) = 0 \qquad \mathrm{M}_\sigma(v) = 0}{\langle (v_0, \tau)\ \&\&\ e, \sigma\rangle \to_e \langle (v, \mathsf{signed\ int})), \sigma\rangle}. \tag{12}$$

$$\frac{\tau - \text{скалярный тип} \qquad \mathrm{M}_\sigma(v_0) \neq 0 \qquad \mathrm{M}_\sigma(v) = 1}{\langle (v_0, \tau)\ ||\ e, \sigma\rangle \to_e \langle (v, \mathsf{signed\ int})), \sigma\rangle}. \tag{13}$$

$$\frac{\tau - \text{скалярный тип} \qquad \Pi_\sigma = \emptyset \qquad \mathrm{M}_\sigma(v) \neq 0}{\langle (v, \tau)\ \&\&\ e, \sigma_0\rangle \to_e \langle \mathbf{OrAnd}(v), \sigma_0\rangle}. \tag{14}$$

$$\frac{\tau - \text{скалярный тип} \qquad \Pi_\sigma = \emptyset \qquad \mathrm{M}_\sigma(v) = 0}{\langle (v, \tau)\ ||\ e, \sigma_0\rangle \to_e \langle \mathbf{OrAnd}(v), \sigma_0\rangle}. \tag{15}$$

$$\frac{\tau - \text{скалярный тип} \qquad \mathrm{M}_\sigma(v_0) = 0 \qquad \mathrm{M}_\sigma(v) = 0}{\langle \mathbf{OrAnd}(v_0, \tau), \sigma\rangle \to_e \langle (v, \mathsf{signed\ int})), \sigma\rangle}. \tag{16}$$

$$\frac{\tau - \text{скалярный тип} \qquad \mathrm{M}_\sigma(v_0) \neq 0 \qquad \mathrm{M}_\sigma(v) = 1}{\langle \mathbf{OrAnd}(v_0, \tau), \sigma\rangle \to_e \langle (v, \mathsf{signed\ int})), \sigma\rangle}. \tag{17}$$

$$\frac{\tau - \text{скалярный тип} \qquad \Pi_\sigma = \emptyset \qquad \mathrm{M}_\sigma(v) \neq 0}{\langle (v, \tau)?\ e_1 \colon e_2, \sigma_0\rangle \to_e \langle (\tau_c) e_1, \sigma_0\rangle}. \tag{18}$$

$$\frac{\tau - \text{скалярный тип} \qquad \Pi_\sigma = \emptyset \qquad \mathrm{M}_\sigma(v) = 0}{\langle (v, \tau)?\ e_1 :\ e_2, \sigma_0\rangle \to_e \langle (\tau_c) e_2, \sigma_0\rangle}. \tag{19}$$

5.2.4. Операции с побочными эффектами

Как и в C, любое изменение содержимого памяти в программе на языке C-light происходит посредством побочных эффектов. Выражение характеризуется в первую очередь своим значением и типом. Поэтому даже простая операция присваивания вызывает побочный эффект. Таким образом, оператор-выражение без побочных эффектов, например `a+b;`, ничего не вычисляет.

Все побочные эффекты, генерируемые при вычислении выражений, помещаются в очередь Π, которая очищается при достижении контрольной точки.

$$\frac{\eta \in \Pi_\sigma \qquad \eta = \text{upd(lval, val)}}{\langle e, \sigma\rangle \to_e \langle e, \sigma(\text{M}(\text{lval} \leftarrow \text{val}), \Pi - \{\eta\})\rangle}. \tag{20}$$

$$\langle (v, \tau) \odot = e_2, \sigma_0\rangle \to_e \langle (v, \tau) = (v, \tau) \odot e_2, \sigma_0\rangle. \tag{21}$$

$$\frac{\langle e_2, \sigma_0\rangle \to_e \langle e, \sigma\rangle}{\langle e_1 \odot = e_2, \sigma_0\rangle \to_e \langle e_1 \odot = e, \sigma\rangle}. \tag{22}$$

$$\frac{\langle (\tau_1)(v_0, \tau_0), \sigma_0\rangle \to_e \langle v, \sigma_0\rangle \qquad \tau - \text{не массив}}{\langle \text{lval} = (v_0, \tau_0), \sigma_0\rangle \to_e \langle v, \sigma_0(\Pi + \{\text{upd}(\text{lval}, v)\})\rangle}. \tag{23}$$

$$\frac{v \neq \Omega}{\langle \text{lval++}, \sigma\rangle \to_e \langle (\text{M}_\sigma(lval), \tau), \sigma(\Pi + \{\text{upd}(\text{lval}, v)\})\rangle}, \tag{24}$$

где $v = \text{BinOpSem}(\sigma, +, (\text{M}_\sigma(\text{lval}), \tau), (v', \mathsf{signed\ int}))$ и $\text{M}_\sigma(v') = 1$.

$$\frac{v = \Omega}{\langle \text{lval++}, \sigma\rangle \to_e \langle \langle \Omega, \sigma\rangle}, \tag{25}$$

где $v = \text{BinOpSem}(\sigma, +, (\text{M}_\sigma(\text{lval}), \tau), (v', \mathsf{signed\ int}))$ и $\text{M}_\sigma(v') = 1$.

5.2.5. Вызовы функций

При вызове функции происходит переход к ее телу. Вызов функции является контрольной точкой, поэтому очередь отложенных побочных эффектов очищается.

$$\frac{\Pi_\sigma = \emptyset \qquad \text{все } e_i \text{ являются значениями}}{\langle (f, \tau)(e_1, ..., e_n), \sigma\rangle \to_e \langle \mathbf{FCall}(f, \tau)(e_1, ..., e_n), \sigma\rangle}, \tag{26}$$

где $\tau = \tau_1 \times ... \times \tau_n \to \tau_r$.

$$\frac{\langle s, \text{InstParms}(\sigma_0, \text{args}, [e_1 ... e_n])\rangle \to_s \langle v_s, \sigma\rangle}{\langle \mathbf{FCall}(f, \tau)(e_1, ..., e_n), \sigma_0\rangle \to_e \langle (v_s, \tau_r), \sigma_0(\text{M} \leftarrow \text{M}_\sigma)\rangle}, \tag{27}$$

где s — тело функции.

5.3. Декларации

Рассмотрим аксиомы и правила для деклараций аннотаций, типов, простых переменных, структур и функций. Инициализатором объекта может быть выражение, поэтому могут срабатывать правила для отношения $\rightarrow_e$. При нормальном завершении будет возвращаться значение VarDeclVal.

$$\langle /\% \ \_condbody \ \%/, \sigma\rangle \rightarrow_v \langle \mathrm{VarDeclVal}, \sigma\rangle. \tag{28}$$

$$\langle \mathsf{typedef}\ \tau\ \mathsf{Id};, \sigma\rangle \rightarrow_v \langle \mathrm{VarDeclVal}, \sigma(\mathrm{TD}(\mathsf{Id} \leftarrow \tau))\rangle. \tag{29}$$

$$\frac{\mathrm{GLF}_\sigma \neq 0}{\langle \tau\ \mathsf{id};, \sigma\rangle \rightarrow_v \langle \mathrm{VarDeclVal}, \sigma(\Gamma(\mathsf{id} \leftarrow \tau), \mathrm{M}(\mathsf{id} \leftarrow \omega))\rangle}. \tag{30}$$

$$\frac{\mathrm{GLF}_\sigma = 0 \qquad \tau - \text{скалярный тип}}{\langle \tau\ \mathsf{id};, \sigma\rangle \rightarrow_v \langle \mathrm{VarDeclVal}, \sigma(\Gamma(\mathsf{id} \leftarrow \tau), \mathrm{M}(\mathsf{id} \leftarrow 0))\rangle}. \tag{31}$$

$$\frac{\begin{array}{c}\langle \tau\ \mathsf{id};, \sigma_0\rangle \rightarrow_v \langle \mathrm{VarDeclVal}, \sigma'\rangle \\ \langle \mathsf{id} = e, \sigma'\rangle \rightarrow_e^* \langle (v, \tau), \sigma\rangle \qquad \Pi_\sigma = \emptyset\end{array}}{\langle \tau\ \mathsf{id} = e;, \sigma_0\rangle \rightarrow_v \langle \mathrm{VarDeclVal}, \sigma\rangle}. \tag{32}$$

$$\frac{\langle \tau\ \mathsf{id};, \sigma_0\rangle \rightarrow_v \langle \mathrm{VarDeclVal}, \sigma'\rangle \qquad \langle \mathsf{id} = e, \sigma'\rangle \rightarrow_e^* \langle \Omega, \sigma\rangle}{\langle \tau\ \mathsf{id} = e;, \sigma_0\rangle \rightarrow_v \langle \Omega, \sigma\rangle}. \tag{33}$$

$$\langle \mathsf{struct\ id;}\ , \sigma\rangle \rightarrow_v \langle \mathrm{VarDeclVal}, \sigma(\Sigma(\mathsf{id} \leftarrow \emptyset))\rangle. \tag{34}$$

$$\langle \mathsf{struct\ id}\{fields\};\ , \sigma\rangle \rightarrow_v \langle \mathrm{VarDeclVal}, \sigma(\Sigma(\mathsf{id} \leftarrow fields))\rangle. \tag{35}$$

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

$$\langle \mathsf{enum\ id}\{decls\};\ , \sigma\rangle \rightarrow_v \langle \mathsf{signed\ int}\ decls, \sigma\rangle. \tag{36}$$

$$\langle \tau\ \mathsf{id}(\tau_1\ v_1, ..., \tau\ v_n);\ , \sigma\rangle \rightarrow_v \langle \mathrm{VarDeclval}, \sigma(\Phi(\mathsf{id} \leftarrow (\tau, \tau_1, ..., \tau_n)))\rangle. \tag{37}$$

$$\langle \varepsilon, \sigma \rangle \to_v \langle \text{VarDeclVal}, \sigma \rangle. \tag{38}$$

Декларации могут следовать одна за другой, и в правилах для блока предполагается, что в начале блока находится последовательность деклараций и будут срабатывать следующие правила:

$$\frac{\langle vd, \sigma_0 \rangle \to_v \langle \text{VarDeclVal}, \sigma' \rangle \qquad \langle vs, \sigma' \rangle \to_v \langle v, \sigma \rangle}{\langle vdvs, \sigma_0 \rangle \to_v \langle v, \sigma \rangle}. \tag{39}$$

$$\frac{\langle vd, \sigma_0 \rangle \to_v \langle \Omega, \sigma \rangle}{\langle vdvs, \sigma_0 \rangle \to_v \langle \Omega, \sigma \rangle}. \tag{40}$$

5.4. Операторы

Пустой оператор

$$\langle ; , \sigma \rangle \to_s \langle \text{OkVal}, \sigma \rangle. \tag{41}$$

Оператор-выражение. Конец любого полного выражения является контрольной точкой, поэтому очередь отложенных побочных эффектов очищается.

$$\frac{\langle e, \sigma_0(\Pi \leftarrow \emptyset) \rangle \to_e^* \langle (v, \tau), \sigma \rangle \qquad \Pi_\sigma = \emptyset}{\langle e; , \sigma_0 \rangle \to_s \langle \text{OkVal}, \sigma \rangle}. \tag{42}$$

$$\frac{\langle e, \sigma_0(\Pi \leftarrow \emptyset) \rangle \to_e^* \langle \Omega, \sigma \rangle}{\langle e; , \sigma_0 \rangle \to_s \langle \Omega, \sigma \rangle}. \tag{43}$$

Передача управления. Проблема заключается в том, что любой оператор передачи управления требует дополнительного прохода по телу программы, чтобы определить следующий фрагмент кода. Поэтому результаты работы этих операторов изображаются в виде специальных значений: GoVal(L), BreakVal, ContVal и RetVal(v_{opt}, τ_{opt}). При переходе на метку L абстрактная машина находит фрагмент программы, помеченный этой меткой; при выходе из функции вычисляется возвращаемое значение, если оно есть, и происходит переход в конец тела функции; значения, получаемые в операторе выхода из блока и перехода на следующую итерацию, отлавливаются в операторах циклов (см. далее).

$$\langle \mathsf{goto}\ \text{L}; , \sigma \rangle \to_s \langle \text{GoVal(L)}, \sigma \rangle. \tag{44}$$

$$\langle \mathsf{break};, \sigma \rangle \to_s \langle \mathrm{BreakVal}, \sigma \rangle. \tag{45}$$

$$\langle \mathsf{continue};, \sigma \rangle \to_s \langle \mathrm{ContVal}, \sigma \rangle. \tag{46}$$

$$\langle \mathsf{return};, \sigma \rangle \to_s \langle \mathrm{RetVal}(\mathsf{void}), \sigma \rangle. \tag{47}$$

$$\frac{\langle e, \sigma_0(\Pi \leftarrow \emptyset) \rangle \to_e^* \langle (v, \tau), \sigma \rangle \qquad \Pi_\sigma = \emptyset}{\langle \mathsf{return}\ e;, \sigma_0 \rangle \to_s \langle \mathrm{RetVal}(v), \sigma \rangle}. \tag{48}$$

Составные операторы. Правила этой группы определяют обработку последовательностей операторов и блоков. Отметим два важных момента. Во-первых, подразумевается стандартное ограничение языка C: никакой оператор в блоке не может предшествовать декларациям. Во-вторых, при входе в блок и выходе из него происходит работа со стеком. В частности, это позволяет корректно обрабатывать локальное переопределение переменных. Однако, поскольку это относится к деталям реализации абстрактной машины, явно стек в правилах не присутствует.

$$\frac{\langle decls, \sigma_0 \rangle \to_v \langle \mathrm{VarDeclVal}, \sigma_1 \rangle \qquad \langle stmts, \sigma_1 \rangle \to_s \langle v, \sigma_2 \rangle}{\langle \{decls\ stmts\}, \sigma_0 \rangle \to_s \langle v, \sigma_0(M \leftarrow M_{\sigma_2}) \rangle}. \tag{49}$$

$$\frac{\langle decls, \sigma_0 \rangle \to_v \langle \Omega, \sigma \rangle}{\langle \{decls\ stmts\}, \sigma_0 \rangle \to_s \langle \Omega, \sigma \rangle}. \tag{50}$$

$$\langle \varepsilon, \sigma \rangle \to_s \langle \mathrm{OkVal}, \sigma \rangle. \tag{51}$$

$$\frac{\langle s_1, \sigma_0 \rangle \to_s \langle \mathrm{OkVal}, \sigma_1 \rangle \qquad \langle stmts, \sigma_1 \rangle \to_s \langle v, \sigma \rangle}{\langle s_1 stmts, \sigma \rangle \to_s \langle v, \sigma \rangle}. \tag{52}$$

$$\frac{\langle s_1, \sigma_0 \rangle \to_s \langle v, \sigma \rangle \qquad v \neq \mathrm{OkVal} \ \text{ или } \ \mathrm{GoVal}}{\langle s_1 stmts, \sigma_0 \rangle \to_s \langle v, \sigma \rangle}. \tag{53}$$

$$\frac{\langle s_1, \sigma_0 \rangle \to_s \langle \mathrm{GoVal(L)}, \sigma_1 \rangle \qquad \langle stmts, \sigma_0 \rangle \to_s^* \langle s_2, \sigma_2 \rangle}{\langle s_1 stmts, \sigma_0 \rangle \to_s \langle s_2, \sigma_1 \rangle}. \tag{54}$$

Условные операторы. Правила для условного оператора if и оператора переключателя switch очевидны: в зависимости от значения условного выражения выбирается та или иная ветвь. Следует отметить, что перед выполнением конкретной ветви очередь отложенных побочных эффектов очищается, так как условное выражение является контрольной точкой.

$$\frac{\langle e, \sigma_0(\Pi \leftarrow \emptyset) \rangle \to_e^* \langle \Omega, \sigma \rangle}{\langle \mathsf{if}(e)\ s_1\ \mathsf{else}\ s_2, \sigma_0 \rangle \to_s \langle \Omega, \sigma \rangle}. \tag{55}$$

$$\frac{\langle e, \sigma_0(\Pi \leftarrow \emptyset)\rangle \rightarrow_e^* \langle (v,\tau), \sigma_1\rangle \qquad \langle s_1, \sigma_1\rangle \rightarrow_s \langle v, \sigma\rangle \quad \tau - \text{скалярный тип} \qquad \mathrm{M}_\sigma(v) \neq 0 \qquad \Pi_{\sigma_1} = \emptyset}{\langle \mathsf{if}(e)\ s_1\ \mathsf{else}\ s_2, \sigma_0\rangle \rightarrow_s \langle v, s\rangle}. \tag{56}$$

$$\frac{\langle e, \sigma_0(\Pi \leftarrow \emptyset)\rangle \rightarrow_e^* \langle (v,\tau), \sigma_1\rangle \qquad \langle s_2, \sigma_1\rangle \rightarrow_s \langle v, \sigma\rangle \quad \tau - \text{скалярный тип} \qquad \mathrm{M}_\sigma(v) = 0 \qquad \Pi_{\sigma_1} = \emptyset}{\langle \mathsf{if}(e)\ s_1\ \mathsf{else}\ s_2, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}. \tag{57}$$

Основным отличием оператора switch от оператора case языка Pascal является то, что в нем фактически нет отдельных ветвей. Все тело этого оператора рассматривается как один оператор, а ветви выделяются подходящей расстановкой операторов break. Поэтому правило перехода к соответствующей ветви просто собирает все операторы от нужной метки до конца в один блок. В п. 3.1 было сформулировано ограничение, позволяющее просто описать эту сборку ветвей в семантике.

$$\frac{\langle e, \sigma_0(\Pi \leftarrow \emptyset)\rangle \rightarrow_e^* \langle \Omega, \sigma\rangle}{\langle \mathsf{switch}(e)\{\mathsf{case}\ c_1\colon\ s_1\ \ldots\ \mathsf{case}\ c_n\colon\ s_n\ \mathsf{default}\colon s_{def}\}, \sigma_0\rangle \rightarrow_s \langle \Omega, \sigma\rangle}. \tag{58}$$

$$\frac{\langle e, \sigma_0(\Pi \leftarrow \emptyset)\rangle \rightarrow_e^* \langle (v,\tau), \sigma_1\rangle \qquad \langle \{s_i\ \ldots\ s_n\ s_{def}\}, \sigma_1\rangle \rightarrow_s \langle v, \sigma\rangle \quad \tau - \text{скалярный тип} \qquad \mathrm{M}_\sigma(v) = c_i \qquad \Pi_{\sigma_1} = \emptyset}{\langle \mathsf{switch}(e)\{\mathsf{case}\ c_1\colon\ s_1\ \ldots\ \mathsf{case}\ c_n\colon\ s_n\ \mathsf{default}\colon s_{def}\}, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}. \tag{59}$$

$$\frac{\langle e, \sigma_0(\Pi \leftarrow \emptyset)\rangle \rightarrow_e^* \langle (v,\tau), \sigma 1\rangle \qquad \langle s_{def}, \sigma_1\rangle \rightarrow_s \langle v, \sigma\rangle \quad \tau - \text{скалярный тип} \qquad \mathrm{M}_\sigma(v) \notin \{c_1, \ldots, c_n\} \qquad \Pi_{\sigma_1} = \emptyset}{\langle \mathsf{switch}(e)\{\mathsf{case}\ c_1\colon\ s_1\ \ldots\ \mathsf{case}\ c_n\colon\ s_n\ \mathsf{default}\colon s_{def}\}, \sigma_0\rangle \rightarrow_s \langle v, \sigma\rangle}. \tag{60}$$

Циклы. В общем случае циклы языка C-light представляют собой значительное препятствие при описании семантики, поскольку в них могут содержаться операторы break и continue. Поэтому операционная семантика непосредственно работает не с исходными циклами, а со специальными конструкциями $\boldsymbol{T}$ и $\boldsymbol{O}$, предложенными в [11]. Конструкция $\boldsymbol{O}$ (loop) получает в качестве аргументов условное выражение и тело цикла и просто моделирует исполнение тела цикла при заданном условии. Конструкция $\boldsymbol{T}$ (trap) получает на входе значение, которое должно быть отловлено, и оператор, который должен быть исполнен. Естественно, отлавливаемыми значениями являются BreakVal и ContVal:

$$\textsf{while}\ (g)\ s \quad \cong \quad \boldsymbol{T}(\text{BreakVal},\ \boldsymbol{O}(g,\ \boldsymbol{T}(\text{ContVal},\ s)))$$

$$\textsf{for}\ (e_1;\ e_2;\ e_3)\ s \quad \cong \quad \{e_1;\ \boldsymbol{T}(\text{BreakVal},\ \boldsymbol{O}(e_2,\{\boldsymbol{T}(\text{ContVal},\ s)\ e_3\}))\}$$

$$\textsf{do}\ s\ \textsf{while}\ (g) \quad \cong \quad \boldsymbol{T}(\text{BreakVal},\ \{\boldsymbol{T}(\text{ContVal},\ s)\ \boldsymbol{O}(g,\ \boldsymbol{T}(\text{ContVal},\ s))\})$$

$$\frac{\langle s, \sigma_0\rangle \to_s \langle v, \sigma\rangle}{\langle \mathbf{T}(v, s), \sigma_0\rangle \to_s \langle \text{OkVal}, \sigma\rangle}. \tag{61}$$

$$\frac{\langle s, \sigma_0\rangle \to_s \langle v, \sigma\rangle \qquad v' \neq v}{\langle \mathbf{T}(v', s), \sigma_0\rangle \to_s \langle v, \sigma\rangle}. \tag{62}$$

$$\frac{\langle e, \sigma_0(\Pi \leftarrow \emptyset)\rangle \to_e^* \langle \Omega, \sigma\rangle}{\langle \mathbf{O}(g, s), \sigma_0\rangle \to_s \langle \Omega, \sigma\rangle}. \tag{63}$$

$$\frac{\begin{array}{c}\langle e, \sigma_0(\Pi \leftarrow \emptyset)\rangle \to_e^* \langle (v, \tau), \sigma\rangle \qquad \tau - \text{скалярный тип} \\ \mathrm{M}_\sigma(v) = 0 \qquad \Pi_{\sigma_1} = \emptyset\end{array}}{\langle \mathbf{O}(g, s), \sigma_0\rangle \to_s \langle \text{OkVal}, \sigma\rangle}. \tag{64}$$

$$\frac{\begin{array}{c}\langle e, \sigma_0(\Pi \leftarrow \emptyset)\rangle \to_e^* \langle (v, \tau), \sigma_1\rangle \qquad \langle s, \sigma_1\rangle \to_s \langle val, \sigma\rangle \\ \tau - \text{скалярный тип} \qquad val \neq \text{OkVal} \qquad \mathrm{M}_\sigma(v) \neq 0 \qquad \Pi_{\sigma_1} = \emptyset\end{array}}{\langle \mathbf{O}(g, s), \sigma_0\rangle \to_s \langle val, \sigma\rangle}. \tag{65}$$

$$\frac{\begin{array}{c}\langle e, \sigma_0(\Pi \leftarrow \emptyset)\rangle \to_e^* \langle (v, \tau), \sigma_1\rangle \qquad \langle s, \sigma_1\rangle \to_s \langle \text{OkVal}, \sigma_2\rangle \\ \langle \mathbf{O}(g, s), \sigma_2\rangle \to_s \langle val, \sigma\rangle \qquad \tau - \text{скалярный тип} \\ \mathrm{M}_\sigma(v) \neq 0 \qquad \Pi_{\sigma_1} = \emptyset\end{array}}{\langle \mathbf{O}(g, s), \sigma_0\rangle \to_s \langle val, \sigma\rangle}. \tag{66}$$

5.5. Свойства семантики

Рассматривая транзитивное рефлексивное замыкание ($\to_s^*$) для отношения перехода, можно определить семантику программы как отображение из *Spaces* в 2^{Spaces}:

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

т. е. семантика рассматривает все возможные исполнения программы, начинающиеся из состояния σ. Отображение $\mathcal{M}$ называется ***семантикой*** для языка C-light. 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 $\langle S, \sigma \rangle \to \langle S_1, \tau \rangle$.

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

6. ЗАКЛЮЧЕНИЕ

Описанный здесь язык C-light лежит в основе проекта, цель которого — разработка системы верификации C-light-программ. Достоинство языка C-light состоит в том, что он составляет представительное подмножество языка C и позволяет работать с динамической памятью. Кроме того, язык C-light существенно расширяет Паскаль, для которого в 1991—1996 гг. была разработана система верификации программ СПЕКТР [3].

Для верификации программ на языке C-light предполагается использовать двухуровневую схему, когда на первом этапе осуществляется трансляция языка C-light в его подмножество C-light-kernel, а на втором — с помощью правил аксиоматической семантики языка C-light-kernel генерируются условия корректности. Цель трансляции языка C-light в язык C-light-kernel — упростить правила аксиоматической семантики, поскольку для языка C-light они достаточно громоздки. В языке C-light-kernel по сравнению с языком C-light будут устранены побочные эффекты в выражениях, некоторые виды циклов, операторы передачи управления break и continue, а оператор switch будет преобразован в более простую форму. Заметим, что эта схема предусматривает также формальное обоснование корректности трансляции языка C-light в язык C-light-kernel и непротиворечивости аксиоматической семантики языка C-light-kernel относительно его операционной семантики.

СПИСОК ЛИТЕРАТУРЫ

1. **Керниган Б., Ритчи Д.** Язык программирования Си. — М.: Финансы и статистика, 1985.
2. **Непомнящий В.А., Рякин О.М.** Прикладные методы верификации программ. — М.: Радио и связь, 1988.

3. **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.
4. **Apt K.R., Olderog E.R.** Verification of sequential and concurrent programs. — Berlin a.o.: Springer Verlag, 1991.
5. **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).
6. **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).
7. **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).
8. **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).
9. **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).
10. **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).
11. **Norrish M.** C formalised in HOL // PhD thes., Computer Lab., Univ of Cambridge, 1998.
12. **Plotkin G.D.** A structure approach to operational semantics: Technical Rep. FN-19, Aarhus University, DAIMI, 1981.

ПРИЛОЖЕНИЕ

СИНТАКСИС ЯЗЫКА C-LIGHT

Грамматика C не является полностью «левосторонней» грамматикой класса LALR(1), хотя и близка к этому. С грамматикой C-light ситуация еще сложнее, так как в аннотациях фрагменты стандартных выражений C беспорядочно «вкраплены» в текст неизвестной структуры. В то же время для описания грамматики хотелось бы использовать некоторую наглядную запись. В результате грамматика C-light записывается в виде формул BNF, которые интерпретируются несколько необычно: при неуспехе разбора фрагмента текста программы возможен откат назад, чтобы проверить другие варианты.

Простейший пример: если в тексте программы определены переменные `pi`, `radius`, `epsilon` и `success`, то начало аннотации

`(pi*radius*radius < epsilon ==> success)`

похоже на выражение языка C, и подвох обнаруживается только после распознавания фрагмента `(pi*radius*radius < epsilon`. В этом случае необходимо выполнить откат назад, предположить, что левая круглая скобка не есть часть выражения C, распознать выражение C `pi*radius*radius < epsilon`, распознать `==>` как фрагмент текста аннотации, распознать `success` как выражение C и т.д.

Использование возвратов приводит к нескольким проблемам.

- Транслятор не умеет читать мысли, даже с откатами назад, поэтому выражения C в аннотациях иногда приходится обрамлять специальными скобками (% и %).
- Постоянные возвраты могут привести к зацикливанию, поэтому **порядок перечисления вариантов** в приведенных здесь правилах **является существенным**; кроме того, правила вида «либо терм, либо пусто» применяются очень осторожно. Эти проблемы хорошо знакомы тем, кто имел дело с Прологом.

Откат назад затрудняет диагностику синтаксических ошибок, поскольку транслятор помнит самую дальнюю точку текста, которую он смог достичь, но не может внятно объяснить, почему он не смог пройти дальше. Как и в Прологе, ситуацию существенно улучшает введение понятие «точки отсечения», т. е. места в правиле, через которое нельзя «пройти назад». Точки отсечения могут быть размещены множеством способов, лишь бы они не влияли на анализ любого грамматически кор-

ректного текста. В приводимых правилах они не указаны, так как не являются частью формального описания грамматики.

Лексические соглашения

Текст программы состоит из **лексем**, разделяемых **пробельными элементами**. Разбиение текста происходит слева направо в соответствии со «стратегией lex».

Пробельными элементами языка являются:

- символы пробела, табуляции и перевода строки;
- комментарии C (/*...*/), причем вложенные комментарии могут быть разрешены или запрещены в зависимости от настроек транслятора;
- комментарии C++ (//...).

C-light не поддерживает непереносимый **token pasting** с использованием пустого комментария /**/. Впрочем, в связи с отсутствием препроцессора ANSI token pasting ## тоже не поддерживается.

Как обычно, пробельные элементы не выделяются внутри символьных и строковых констант. Кроме того, поддерживается перенос строковых констант со строки на строку при помощи обратной косой черты перед символами перевода строки.

Имеется одно исключение из обычных правил: строки вида /*%...%*/ являются не комментариями, а аннотациями; /*% и %*/ есть просто другое написание для /% и %/.

Все лексемы C-light могут быть разбиты на пять категорий:

- пунктуаторы, в том числе:
  - знаки операций,
  - разделители,
  - парные скобки,
  - ключевые слова;
- идентификаторы;
- численные константы, в том числе:
  - десятичные, восьмеричные и шестнадцатеричные целые,
  - отсутствующие в ANSI C двоичные целые, записываемые как префикс 0b, за которым следуют цифры 0 или 1,
  - рациональные числа с указанием и без указания экспоненты;
- символьные константы, однобайтовые и мультибайтовые;
- строковые константы, трактуемые как массивы однобайтовых символов, завершающиеся байтом 0.

Синтаксис лексем определен в соответствии с синтаксисом ANSI C со следующими исключениями:

1. Добавлены парные скобки /%...%/ и /*%...%*/, предназначенные для обрамления аннотаций.
2. Добавлены парные скобки (%...%), предназначенные для явного обрамления выражений C-light в аннотациях.
3. В качестве «монолитных» лексем добавлены цепочки символов ==>, <==, =−> и <−=, предназначенные для использования в аннотациях, например для записи импликаций.
4. Для использования в аннотациях добавлены символы @, ‘, $ и $$.
5. Целочисленные константы могут быть снабжены суффиксами l или L для указания модификатора длины long, суффиксами s или S для указания модификатора длины short, суффиксами u или U для указания модификатора unsigned. Регистр и порядок следования этих суффиксов значения не имеют.
6. В числе с плавающей точкой должно быть минимум по одной цифре перед десятичной точкой и после нее.
7. Однобайтовая символьная константа трактуется как значение типа char, а не как int, т. е. применяется правило ANSI C++.
8. В символьных и строковых константах разрешено использование не только обычных escape-последовательностей, таких как \n, \10, \012 или \x0a, но и escape-последовательностей с явно указанными размерностями 10 (\0d10) и 2 (\0b1010). Это расширение сделано для облегчения автоматической генерации некоторых текстов программ.

Правила верхнего уровня

__external_decln:
    enum_decln
    struct_decln
    linkage_specification
    typedef_decln
    _cond_control
    _cond
    ;

__function_body:
    statement

Аннотации

_ cond:
/% *_ condbody* %/

_ cond_ control:
/% (inline | extern | auto) %% *function* %/

_ condbody:
(*expn* | (% expn %) | [*_ condbody*] | { *_ condbody* } |
(*_ condbody*) | *не скобка и не %%*)*

Имена

class_ name:
IdLexem

field_ name:
*objname /*declared before*/*

objname:
:: *objname*
class_ name :: *objname*
IdLexem

simple_ type_ name:
qualified_ class_ name
qualified_ enum_ name
std_ type
IdLexem */*name of semi-defined typedef*/*
IdLexem */*name of class-argument of template*/*

type_ name:
type_ spec abs_ drator$_{opt}$

conv_fname:
type_ spec pointer_ operator$_{opt}$

qualified_class_name:
:: *qualified_class_name*
class_name :: *qualified_class_name*
class_name

qualified_enum_name:
:: *qualified_enum_name*
class_name :: *qualified_enum_name*
IdLexem /**name of declared enumeration**/

Декларации

data_decln:
dcl_spec type_spec (*drator data_init* (, *drator data_init*)*)$_{opt}$;

data_init:
(= *ass_expn* | = { *data_init_list* } |
(*ass_expn* (, *ass_expn*)*))$_{opt}$

data_init_list:
(*const_expn* | { *data_init_list* })(, (*const_expn* |
{ *data_init_list* }))*

dcl_spec:
(*storage_class_spec* | *cv_qualifier* | *function_spec*)*

type_spec:
cv_qualifier_list (*simple_type_name* | *struct_spec* |
((*class_key* | enum) IdLexem))

drator:
drator_head drator_tail

drator_head:
objname /**maybe, not declared before**/
pointer_operator drator_head
call_framing drator_head
(*drator*)

drator_tail:
(*drator_tail_func* | [*const_expn*$_{opt}$])*

drator_ tail_ func:
((... | void | (*argument_ decln* (, *argument_ decln*)* (,...)$_{opt}$)$_{opt}$)

argument_ decln:
dcl_ spec type_ spec (*drator* | *abs_ drator*)$_{opt}$ (= *ass_ expn*)$_{opt}$

abs_ drator:
pointer_ operator abs_ drator$_{opt}$
(*abs_ drator*) *drator_ tail*
call_ framing abs_ drator$_{opt}$
drator_ tail

struct_ decln:
class_ key class_ memmodel rtti IdLexem ;

struct_ spec:
class_ key class_ memmodel rtti class_ name {
member_ list *_ cond*$_{opt}$ }

typedef_ decln:
typedef *dcl_ spec type_ spec drator* ;

enum_ decln:
enum IdLexem$_{opt}$ { (IdLexem (= *const_ expn*)$_{opt}$
(, IdLexem (= *const_ expn*)$_{opt}$)*)$_{opt}$ }

function:
dcl_ spec (*type_ spec drator_ head* | *drator_ head*)
drator_ tail_ func (; | { __function_body })

linkage_ specification:
extern LiteralLexem { (*linkage_ specification*)+ }$_{opt}$
extern (LiteralLexem /* *"C"or "Pascal"* */)$_{opt}$ (*function* |
data_ decln)

member_ list:
(*member_ decln* (: *const_ expn*)$_{opt}$ | : *const_ expn* |
access_ spec :)*

member_ decln:
qualified_ class_ name ;
enum_ decln
function ;$_{opt}$

data_ decln
typedef_ decln
friend (*struct_ decln* | *function* $;_{opt}$ | *data_ decln*)

pointer_ operator:
* *cv_ qualifier_ list*
& *cv_ qualifier_ list*

function_ spec:
inline
virtual

rtti:
__rtti$_{opt}$

std_ type:
std_ type_ sizer std_ type_ signer$_{opt}$ *std_ type_ key*$_{opt}$
std_ type_ signer std_ type_ sizer$_{opt}$ *std_ type_ key*$_{opt}$
std_ type_ key
void

std_ type_ key:
void
bool
char
wchar_t
int
float
double

std_ type_ signer:
signed
unsigned

std_ type_ sizer:
short
long

storage_ class_ spec:
auto

register
static
extern

access_spec:
private
protected
public

class_key:
class
struct
union

class_memmodel:
(near | far | huge near$_{opt}$ | __except | _import)$_{opt}$

cv_qualifier_list:
(const | volatile)*

call_framing:
cdecl
pascal
interrupt
near
far
huge

Выражения

expn:
ass_expn (, *ass_expn*)*

const_expn:
cond_expn /* *without variables or calls of functions* */

ass_expn:
unary_expn ass_operator ass_expn
cond_expn

ass_operator: one of

= *= /= %= += −−=
<<= >>= &= ^= |=

cond_expn:

ration_expn (? *expn* : *cond_expn*)$_{opt}$

ration_expn:

cast_expn (*ration cast_expn*)*

ration: one of

.* −>* * / % + −− << >> <
> <= >= == != & ^ | && ||

cast_expn:

(*type_name*) ((*cast_expn*) | *cast_expn*)
unary_expn

unary_expn:

postfix_expn
++ *unary_expn*
−− *unary_expn*
unary_operator cast_expn
allocation_expn
deallocation_expn
sizeof *unary_expn*
sizeof (*type_name*)

unary_operator: one of

& * + − ~ !

allocation_expn:

new ((*expn*))$_{opt}$ ::$_{opt}$(*type_spec* ((IdLexem)*
([*expnopt*])*)$_{opt}$ | (*type_name*)) *data_init*

deallocation_expn:

::$_{opt}$ delete ([*expn*$_{opt}$])$_{opt}$ *cast_expn*

postfix_expn:

primary_expn ([*expn*] | (*ass_expn*$_{opt}$ (, *ass_expn*)*) |
. *field_name* | −> *field_name* | ++ | −−)*

primary_expn:

ConstLexem
LiteralLexem
this
true
false
(*expn*)
(*simple_type_name* | (*simple_type_name*) |
qualified_class_name :: conv_fname | *conv_fname* |
(const_cast | dynamic_cast | reinterpret_cast |
static_cast) < *type_name* >
) ($expn_{opt}$)
typeid ((*type_name* | *expn*))
objname /**declared before**/

Операторы

statement:

{ $\_cond_{opt}$ (*statement*)* $\_cond_{opt}$ }
(IdLexem /**label name**/ | case *const_expn* | default) : *statement*
expn_statement
selection_statement
iteration_statement
jump_statement
enum_decln
data_decln

expn_statement:

$expn_{opt}$;

iteration_statement:

while (*expn*) *statement*
do *statement* while (*expn*) ;
for ((*expn_statement* |*data_decln*) $expn_{opt}$; $expn_{opt}$) *statement*

jump_statement:

(goto IdLexem /* *label name* */ | continue | break | return $expn_{opt}$) ;

selection_statement:

if (*expn*) *statement* (else *statement*)$_{opt}$
switch (*expn*) *statement*

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

НА ПУТИ К ВЕРИФИКАЦИИ С-ПРОГРАММ. ЧАСТЬ 1. ЯЗЫК С-LIGHT

Препринт
84

Рукопись поступила в редакцию 15.01.2001
Рецензент Ф. А. Мурзин
Редактор Л. А. Карева

Подписано в печать 05.02.2001
Формат бумаги 60×84 1/16 Объем 2,7 уч.-изд.л., 3,0 п.л.
Тираж 50 экз.

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