УДК 519.681.3

НА ПУТИ К ВЕРИФИКАЦИИ С-ПРОГРАММ. ЯЗЫК C-LIGHT И ЕГО ТРАНСФОРМАЦИОННАЯ СЕМАНТИКА

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

Институт систем информатики им. Академика А.П. Ершова СО РАН
630090, Россия, г. Новосибирск, пр. Академика Лаврентьева, 6
факс: 332 3494, тел.: 330 8652, e-mail: {vnep, anureev, promsky}@iis.nsk.su

Предложен ориентированный на верификацию язык C-light, который является представительным подмножеством языка С. Важные отличительные черты языка C-light — это детерминированная семантика выражений, ограниченное использование операторов `switch` и goto, и применение операций `new` и `delete` языка C++ для работы с динамической памятью вместо библиотечных функций. Для верификации C-light программ применяется двухуровневый подход, включающий этапы трансляции языка C-light в его ядро — язык C-kernel и генерации условий корректности с помощью аксиоматической семантики C-kernel. Описаны правила перевода из языка C-light в язык C-kernel и метод формального обоснования их корректности.

A verification-oriented language C-light which is a representative subset of the language C is proposed. Essential features of C-light are deterministic semantics of expressions, restricted use of statements `switch` and goto, the use of operators `new` and `delete` of the language C++ to handle with dynamic memory instead of the use of appropriate library functions. We suggest two-level approach to C-light program verification which includes translation from C-light to its kernel called C-kernel and generation of verification conditions by axiomatic semantics of C-kernel. Rules for translation from C-light to C-kernel and a method of formal justification of their correctness are presented.

Введение

Формальная верификация программ — актуальное направление современного программирования. Особый интерес представляет верификация программ, написанных на распространенных языках системного программирования, таких как С и С++. Трудности верификации программ на языке С подробно обсуждались в [1–3, 10, 11, 17, 18]. Основной проблемой является отсутствие формальной семантики для полного языка С, соответствующего стандарту ANSI [7]. Отметим, что формальная семантика для довольно представительного подмножества С была предложена в [12, 15].

Данный подход к верификации С-программ [1–3, 16] состоит в следующем. Выделено представительное подмножество языка С, названное C-light. Это одно из наиболее обширных подмножеств языка С. Оно содержит все множество операторов языка С при некоторых семантических ограничениях и большинство типов и операций. Отличительной чертой языка C-light является детерминированная семантика выражений. Вместо библиотечных функций для работы с динамической памятью используются операции `new` и `delete` языка С++. В качестве формального определения языка C-light была разработана его полная структурная операционная семантика [1, 3] в стиле Плоткина [19].

При верификации программ вместо операционной семантики обычно используется аксиоматическая семантика в стиле Хоара. Эта семантика более высокого уровня, что позволяет существенно упростить доказательство корректности программ. Непосредственное определение аксиоматической семантики для C-light будет весьма громоздким, что усложнит верификацию. Выходом стало применение двухуровневой схемы верификации C-light-программ. В соответствии с ней в языке C-light выделяется ядро, «хорошее» с точки зрения аксиоматической семантики, в которое транслируются исходные программы. Для этого ядра, названного C-kernel, была разработана аксиоматическая семантика и доказана ее непротиворечивость относительно операционной [2].

Двухуровневая схема верификации предусматривает также формальное определение правил перевода из языка C-light в язык C-kernel и доказательство их корректности. Понятно, что эта задача является трудной. Отметим, что системы эквивалентных преобразований, упрощающих выражения языка С с побочными эффектами, изучались в [10]. Однако, как справедливо отмечалось в [18], доказательство корректности преобразований не описано.

В статье описаны правила перевода из языка C-light в язык C-kernel и метод формального обоснования их корректности. В разд. 1 описан язык C-light. Разд. 2 посвящен обзору абстрактной машины, на которой базируется операционная семантика языка C-light. Обзор языка C-kernel дан в разд. 3. Система правил перевода из языка C-light в язык C-kernel приводится в разд. 4. Правила разбиты на группы в соответствии со стратегией их применения. Там же рассмотрены некоторые свойства правил перевода. В разд. 5 приведены основные теоремы о корректности этой системы правил перевода. В разделе 6 обсуждаются достоинства и перспективы подхода к верификации С-программ.

Работа частично поддержана грантом РФФИ 04-01-00114а.

1. Язык C-light

Обзор языка C-light носит в основном характер сравнения со стандартом C99. Для его описания выделены следующие синтаксические и семантические элементы: *типы*, *декларации*, *выражения*, *операторы* и *программы*.

Типы. Допустимыми типами языка C-light являются все типы языка C за исключением комплексных типов и объединений, при этом используем следующие ограничения и соглашения:

1) перечисления вводят не новые типы, а наборы их констант `signed int`;
2) значения указателей — неинтерпретируемые символьные литералы. Допустимо преобразование целого числа к типу указатель, но не наоборот;
3) неполные типы массивов разрешены только как типы аргументов функций;
4) запрещены функции с переменным числом аргументов;

Декларации.

Множественные неокончательные определения (tentative definitions) для одного идентификатора запрещены, поскольку требуют понятия композитных типов.

Запрещены абстрактные декларации аргументов. Использование спецификатора класса памяти `static` в размере массива, являющегося параметром функции, запрещено. Пустой список аргументов не разрешен — требуется явный `void`.

Функция `main` может быть объявлена только как

```
int main(void){  /* ... */  }
```

Поскольку язык C-light ориентирован на фиксированные (hosted) окружения исполнения [7, § 5.1.2], передача параметров программы в функцию `main` не поддерживается, так как строки, передаваемые через `argv`, зависят от реализации.

Устаревший синтаксис определений функций (в стиле K&R) не поддерживается - требуются прототипы.

Выделенные инициализаторы не поддерживаются.

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

Наконец, имена *всех* статических объектов в программе уникальны, т.е. не только глобальные объекты, но и имена переменных, объявленных в телах функций с ключевым словом `static`. Заметим, что это ограничение не конфликтует с предопределенным идентификатором `__func__`, поскольку он всегда неявный и вне функций не определен.

Выражения. Первое принципиальное отличие выражений языка C-light от C99 состоит в том, что строго фиксирован порядок вычисления выражений. Второе заключается в немедленном срабатывании побочных эффектов без откладывания до ближайшей контрольной точки. Работы других исследователей показывают, что если полностью рассматривать все пути вычисления и стратегии срабатывания побочных эффектов, то осмысленный результат будут иметь только выражения из очень ограниченного класса [18].

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

Составные литералы разрешены только в качестве инициализаторов в декларациях.

Приведения типов указателей ограничено случаем приведения от типа `void*` к произвольному *T*`*`.

Для работы с динамической памятью используются операции `new` и `delete`, которые частично соответствуют аналогичным операциям из языка C++. Связано это с тем, что стандартная библиотека языка C не поддерживается, а выделение в семантике отдельных ее функций приведет к неоднородности. Более того, ничто не мешает пользователю определить свои функции с именами `malloc`, `free` и др. С другой стороны, поддержка библиотеки ничего не даст, поскольку семантика вызова функции состоит в переходе к телу функции, а на самом нижнем уровне библиотечные средства работы с памятью реализованы на ассемблере. Поэтому наиболее простой способ — это введение средств работы с памятью в ядро языка в виде операций.

Операторы. Ограничений на операторы два. Во-первых, все `case`-метки в операторе `switch` 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;}.
```

Во-вторых, 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.

Программы. Исходный текст программы есть последовательность внешних деклараций. На внешнем уровне можно объявить тип (`typedef`-декларации), функцию (прототип или определение), структуру и данные. Кроме обычных конструкций C на верхнем уровне могут присутствовать и аннотации, подробно рассмотренные в [3].

В языке C-light препроцессор отсутствует, поскольку исходная программа препроцессируется до начала ее верификации. Также, по сравнению с [18], не предусмотрена никакая предварительная компиляция.

В процессе исполнения в операционной семантике модульность не поддерживается. Поэтому любая исходная C-light программа состоит из одного файла. В противном случае нам пришлось бы в семантике моделировать компоновку, которая на уровне исходных текстов, а не объектных файлов, сделает семантику слишком громоздкой. Таким образом, если возникнет задача применить операционную семантику языка C-light к программе, весь исходный проект должен быть собран в один файл. С другой стороны, это ограничение не является существенным в двухуровневом подходе. Ведь при верификации в аксиоматической семантике функции заменяются в выводе своими спецификациями. Грубо говоря, зная спецификации вызываемой функции, в логике Хоара не требуется доступ к телу функции. Например, библиотеки пользователя или стандартная библиотека C верифицируются отдельно от программы, которая их использует.

2. Обзор абстрактной машины языка C-light

Операционная семантика определяет процесс исполнения программы в терминах изменений состояний абстрактной машины и, возможно, в терминах взаимодействия с внешним окружением. Уровень детализации состояния может быть различным, но, очевидно, пригодным для классических работ [8, 9], слишком абстрактен для языка C (даже ограниченного до C-light). Мы также не рассматриваем стандартную библиотеку языка C и вызовы системных функций при отсутствии их исходных текстов, поэтому взаимодействие с внешним окружением не моделируется.

В абстрактной машине используются специальные типы: Ctypes — это объединение всех допустимых типов языка C-light; Nat — натуральные числа; Names — множество имен (идентификаторов), встречающихся в программе; Locations — множество неинтерпретируемых констант, соответствующих адресам объектов. Любая такая константа представляет собой пару — строковый литерал и натуральное число или одно из специальных значений, которые будут рассмотрены ниже; TypeSpecs — множество абстрактных имен типов [7, § 6.7.6]. Заметим, что типы Names, Locations и TypeSpecs не входят в CTypes, поскольку в языке C нет встроенного строкового типа; *lv* —специальный модификатор объектных типов, связанный с *l*-значениями [7, § 6.3.2.1].

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

1) MeM — переменная типа (Names × Nat) → Locations, т.е. адресация объектов с учетом уровня их вложенности;
2) MD — переменная типа Locations → CTypes. Это отображение определяет значения, хранимые в памяти;
3) MB — переменная типа Locations × (Nat ∪ Names) → Locations, которая определяет ячейки составных типов.
4) Γ — переменная типа (Names × Nat) → TypeSpecs. Это отображение определяет тип, с которым был объявлен идентификатор на соответствующем уровне вложенности;
5) STD — переменная типа Names → TypeSpecs, т.е. информация о тегах и синонимах типов (`typedef`-декларации);
6) GLF — переменная типа Nat, определяющая текущий уровень вложенности;
7) Val — переменная типа (CTypes × TypeSpecs) ∪ Names, в которой хранится либо значение в паре с его типом, либо одно из специальных исключительных значений.

Определение. *Конфигурация* абстрактной машины языка C-light — это пара $\langle S, \sigma \rangle$, где S — программный фрагмент и σ — состояние.

Произвольная аксиома операционной семантики выглядит как

$$\langle S, \sigma \rangle \to \langle S', \sigma' \rangle .$$

Эта запись означает, что один шаг исполнения программного фрагмента S, начинающийся в состоянии σ, приводит в состояние σ' и S' — тот фрагмент исходной программы, который остается для исполнения. Произвольное правило семантики имеет вид

$$\frac{P_1 \ldots P_n}{\langle S,\sigma\rangle \to \langle S',\sigma'\rangle}.$$

Это означает, что при выполнении условий P_1, …, P_n можно перейти от первой конфигурации ко второй.

Таким образом, исполнение программ в операционной семантике приводит к цепочкам конфигураций, которые в общем случае могут быть и бесконечными:

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

Конфигурация $\langle S_1,\sigma_1\rangle$ называется *начальной*. Если же цепочка конечна и $\langle S_n,\sigma_n\rangle$ — последняя конфигурация в этой цепочке, то говорим, что исполнение фрагмента S_1, начинающееся в состоянии σ_1, приводит в *заключительную* конфигурацию $\langle S_n,\sigma_n\rangle$.

Пустой фрагмент обозначим символом ε. Пустым фрагментом может быть как пустая программа, так и пустое выражение. Обновлением $\sigma(x \leftarrow e)$ состояния σ называется состояние σ', которое совпадет с состоянием σ всюду, за исключением, быть может, переменной x, и $\sigma'(x) = e$.

Задавая транзитивное рефлексивное замыкание →* отношения перехода →, определим семантику C-light программы S как отображения из множества всех состояний Σ в 2^{Σ}:

$$M[S](\sigma) = \{\sigma' \mid \langle S,\sigma\rangle \to^* \langle \varepsilon,\sigma'\rangle\} \cup \{\sigma'(\mathrm{Val} \leftarrow (v,\tau)) \mid \langle S,\sigma\rangle \to^* \langle (v,\tau),\sigma'\rangle\}.$$

3. Обзор языка C-kernel

Язык C-kernel является промежуточным языком двухуровневой схемы верификации программ, в который транслируются программы на языке C-light. Рассмотрим его основные отличия от C-light. Это позволит пояснить причины введения того или иного правила перевода.

Синтаксис лексем языка C-kernel совпадает с синтаксисом лексем языка C-light, но множество ключевых слов сокращается, и остаются следующие: `auto`, `_Bool`, `char`, `delete`, `double`, `else`, `enum`, `float`, `goto`, `if`, `int`, `long`, `new`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `typedef`, `unsigned`, `void`, `while`.

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

Все инициализаторы составных объектов находятся в полноскобочной форме.

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

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

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

Базовым в определении выражений языка C-kernel является понятие *нормализованного выражения*. Это произвольное выражение, построенное по обычным синтаксическим правилам языка C, но с использованием только первичных, унарных и бинарных операций языка C-kernel, а также операции приведения типа. Иными словами, в нормализованных выражениях не происходит изменения содержимого памяти.

Наконец, произвольное выражение языка C-kernel — это одно из следующих выражений:

- $f(v_1, ..., v_n)$, где каждое v_i — это константа или переменная, а f — идентификатор;
- e = $f(v_1, ..., v_n)$, где e — нормализованное выражение, а вызов функции подчинен ограничениям из предыдущего пункта;
- e = `new` τ, где e — нормализованное выражение, а τ — объектный тип;
- e_1 = `new` τ`[`e_2`]`, где e_1 и e_2 — нормализованные выражения;
- e_1 = e_2, где e_1 и e_2 — нормализованные выражения;
- `delete` e, где e — нормализованное выражение;
- `delete []` e, где e — нормализованное выражение.

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

1) оператор вычисления выражения (возможно, пустой);
2) условный оператор `if` с обязательной ветвью `else` (возможно, пустой);
3) оператор цикла `while`, условием которого может быть либо константа, либо переменная;
4) оператор передачи управления `goto`;
5) оператор возврата значения return, причем возвращаемым выражением может быть либо пустым, либо константой, либо переменной;
6) составной оператор (блок).

4. Перевод из языка C-light в язык C-kernel

Правила разбиты на группы в соответствии с их предназначением переписывания: деклараций, операторов и выражений. В некоторых группах правила в свою очередь разбиты на правила нормализации, декомпозиции и правила элиминации. В соответствии с названием первые приводят конструкции к некоторому каноническому виду, вторые проводят декомпозицию сложных конструкций, а третьи удаляют конструкции, которых нет в языке C-kernel.

Для краткости мы не описываем здесь правила перевода для деклараций. Заметим лишь, что их действие состоит в добавлении явных спецификаторов класса памяти (`auto`, `static`), в разбиении списка деклараторов, в приведении инициализаторов к нормальной (полноскобочной) форме и во введении явных тегов для структур. Полная система правил перевода дана в [3].

При описании правил используется обозначение $A \Rightarrow B$, означающее, что фрагмент вида A заменяется на фрагмент вида B.

Нормализация операторов. Правила нормализации приводят операторы к виду, удобному для применения правил из остальных групп. Расставляются дополнительные фигурные скобки, упрощаются условия операторов выбора и циклов и т.д. Например:

если S — это оператор, не являющийся блоком, то

$$\text{while(e) S} \Rightarrow \text{while(e)\{ S \}.} \quad (1)$$

Аналогично превращаются в блоки тела циклов do и for.

Если S — оператор и T не начинается с else, то

$$\text{if(e) A B} \Rightarrow \text{if(e) A else \{\}; B.} \quad (2)$$

Если A не содержит default на верхнем уровне, то

$$\text{switch(e)\{ A \}} \Rightarrow \text{switch(e)\{ A default: ; \}.} \quad (3)$$

Если B не является пустым оператором либо оператором goto, то

$$\text{switch(e)\{ A default: B \}} \Rightarrow \quad (4)$$

$$\text{switch(e)\{ A L: B break; default: goto L; \}.}$$

Если e — не нормализованное выражение, то

$$\text{if(e) A} \Rightarrow \text{\{ auto t x; x = e; if(x) A \},} \quad (5)$$

где x — новая переменная, а t — тип выражения e. Аналогично выносится сложное выражение для оператора switch.

Если A не содержит continue, то

$$\text{while(e)\{ A \}} \Rightarrow \text{while(1)\{ if(e)\{\} else break; A \}.} \quad (6)$$

Элиминация операторов. В каждом правиле данной группы L — это новая уникальная метка, отсутствующая в исходной программе.

Если x — переменная, A пуст либо содержит case, B не содержит case на верхнем уровне, а к C неприменимо правило (4), то

$$\text{switch(x) \{ A case v: B default: C \}} \Rightarrow \quad (7)$$

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

Если A не содержит case и break, а B не содержит break, то

$$\text{switch(x) \{ A default: B \}} \Rightarrow \text{\{ B goto L; A \} L.} \quad (8)$$

Если A не содержит continue, то

do A while(e) ⇒ (9)

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

for(e1; e2; e3) { A } ⇒ { e1; while(e2){ A e3 } }. (10)

Все операторы передачи управления преобразуются к goto. Например:

switch(e) { A break; B } ⇒ (11)

{ switch(e) { A goto L; B } L: };

while(e) { A break; B } ⇒ (12)

{ while(e) { A goto L; B } L: };

while(e) { A continue; B } ⇒ (13)

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

Правила элиминации для выражений. Переписывание выражений сводится к замене каждого выражения, в котором содержатся несколько «подозрительных» подвыражений, на цепочку последовательно вычисляемых выражений, в которой эти подвыражения оказываются вынесенными на верхний уровень. К «подозрительным» будем относить выражения с возможными побочными эффектами либо выражения, в которых не все операнды вычисляются.

e||e' ⇒ (e?1:e') (14)

e&&e' ⇒ (e?e':0) (15)

!e ⇒ (e?0:1) (16)

e_1, ..., e_n; ⇒ { e_1; ... e_n; } (17)

e_1?e_2:e_3; ⇒ { x = e_1; if(x) { e_2; } else { e_3; } } (18)

++e ⇒ (e += 1) (19)

Необходимость в переписывании условных операций обусловлена известным свойством неполного вычисления их операндов.

Операции постфиксного инкремента/декремента переписываются с помощью указателей:

e++ ⇒ (q = &e, y = *q, *q = *q + 1, y) (20)

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

e op e' ⇒ (x = e', y = &e, *y = *y op' x) (21)

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

Правило декомпозиции выражений. Декомпозиция выражений состоит в превращении всех аргументов в вызовах функций и стандартных операций в константы или переменные:

Если e_i — не переменная и не константа, e_{i+1}, ..., e_n — переменные или константы, f — функция или одна из операций +, -, *, /, <, >, <=, =>, !=, ==, то

f(e_1, ..., e_{i-1}, e_i, e_{i+1}, ..., e_n) ⇒ (22)

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

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

Правила нормализации выражений. Если e не имеет вид z или *z для некоторой переменной z, то

e = e' ⇒ (x = e', y = &e, *y = x) (23)

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

*&e ⇒ e (24)

&*e ⇒ e (25)

Если x имеет вид z или *z для некоторой переменной z, то

$$x = e?e_1:e_2 \Rightarrow e?(x = e_1):(x = e_2) \quad (26)$$

$$x = e_1, ..., e_n \Rightarrow (e_1, ..., x = e_n) \quad (27)$$

Если e не аргумент вызова функции и не операнд какой-либо неодноместной операции, то

$$(e) \Rightarrow e \quad (28)$$

В заключение раздела отметим очевидные свойства данной системы правил перевода. Во-первых, правила для переписывания выражений соответствуют ассоциативности и приоритетам операций языка C. Поэтому в результате переписывания фрагмента

```
a = a + 32760 + b + 5
```

результат будет таким же, как для выражения

```
a = (((a + 32760) + b) + 5);
```

что позволяет не рассматривать в правилах вопросы, связанные с переполнениями. Во-вторых, в результате перевода синтаксис выражений в выходной программе соответствует некоторой трехадресной машине — в любом операторе-выражении, кроме операции присваивания, может быть не более одной двухместной операции. Тем самым побочные эффекты становятся единичными: в одном полном выражении содержимое памяти может быть изменено не более одного раза. Все выражения вычисляются полностью. Набор операторов, остающихся в языке C-kernel, семантически соответствует аналогичным операторам языка Pascal.

5. Корректность перевода из C-light в C-kernel

Для упрощения формулировки основных теорем корректности расширим понятие программы.

Определение. *Программа* — это либо последовательность внешних деклараций с одной выделенной функцией — `main`, либо определение одной функции.

Согласно правилам языка C и, следовательно, C-light работа программы начинается с передачи загрузчиком управления функции `main` и заканчивается выходом из нее при отсутствии аварийного останова. Формальная семантика должна следовать этому правилу, поэтому верифицируемая полноценная программа должна иметь именно такой вид. Но вообще говоря, исходя из операционной семантики, входной программой может быть что угодно: один оператор, одно выражение либо несколько операторов. Часто необходимо верифицировать не полную программу, а некоторую функцию. Для того чтобы не приходилось вводить в такой ситуации функцию `main`, понятие допустимой программы расширяется отдельными функциями.

Определение. Множеством *семантических* объектов программы S называется множество Obj(S) идентификаторов функций и всех статических переменных этой программы.

Определение. Множеством *семантических* объектов функции f называется множество Obj(f) идентификаторов автоматических и статических переменных функции f.

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

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

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

$$\text{Attrib}(id) = \text{Attrib}(\phi(id)). \quad (*)$$

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

Важным свойством этого определения является то, что отображение не обязательно связывает только одноименные объекты. Тем самым можно сравнивать программы, полученные не только добавлением новых объектов, но и переименованием старых. Очевидно, что существование такого отображения еще не означает, что программы реализуют одну и ту же вычислимую функцию. Более того, отображений, удовлетворяющих свойству (∗), может быть несколько. Но если среди таких отображений есть хотя бы одно такое, что при любых входных данных результаты выполнения программ S и T совпадают на объектах, связанных этим отображением, то относительно этих объектов программы эквивалентны. Определим формально такую «локальную» эквивалентность.

Определение. Программу T называем *семантическим расширением* программы S, если существует всюду определенное взаимно-однозначное отображение ϕ: Obj(S) → Obj(T) такое, что

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

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

Рассмотрим это определение подробнее. Первое условие означает, что программа T объектно расширяет S, т.е. ϕ — связывание объектов на глобальном уровне. Второе условие означает, что результаты выполнения программ на семантических объектах, связанных отображением ϕ, совпадают. В случае, когда S и T — функции, этого условия достаточно для утверждения, что они эквивалентны. Однако для обычных программ рассматривать окончательные результаты их исполнений не достаточно. Ведь в этом случае рассматриваться будут только статические объекты, а любые автоматические при выходе из блока (и соответственно из функции `main`) исчезают. Тогда возможна следующая ситуация: в программах S и T есть одинаковые статические объекты, которые вообще не изменяются в ходе работы (т.е. используются как константы), но относительно своих автоматических объектов программы реализуют разные функции. Но на выходе из программ сравнить можно будет только статические объекты, и программы будут эквивалентны. Поэтому необходимо рассматривать и некоторые промежуточные результаты. Третье условие делает это определение индуктивным. В нем говорится, что надо рассмотреть и все пары функций, которые отождествляются при отображении ϕ на глобальном уровне. Здесь важно то, что к множествам семантических объектов функций добавляются множества глобальных объектов. Ведь все функции в программе вызываются не в произвольных условиях, а в контексте, формируемом глобальными декларациями программы. Естественно, что сужение отображения $\phi_{(f_1,f_2)}$ для функций f_1 и f_2 на множество Obj(S) должно совпадать с ϕ.

Отношение семантического расширения обозначаем как $S \le_{op} T$. Оно рефлексивно и транзитивно, но не симметрично, поскольку в определении предполагается, что семантические множества второй программы, как правило, шире соответствующих множеств из первой. Специфика правил перевода такова, что добавляются только автоматические объекты. Поэтому если считать, что множества статических объектов обеих программ равномощны, и не рассматривать в заключительных состояниях функций локальные объекты, то появится симметричность, а отношение станет эквивалентностью.

Обозначим множество статических объектов произвольной функции f как Sobj(f).

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

1) $S \lhd_\phi T$,
2) $\forall\sigma\ \phi(M[S](\sigma)) = M[T](\sigma)\big|_{\phi(\mathrm{Obj}(S))}$,
3) для каждой пары функций $f_1 \in S$ и $f_2 \in T$ таких, что $\phi(f_1) = f_2$, существует всюду определенная биекция
$$\phi_{(f_1,f_2)} : \mathrm{Sobj}(f_1) \cup \mathrm{Obj}(S) \to \mathrm{Sobj}(f_2) \cup \mathrm{Obj}(T)$$

такая, что

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

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

(c) для любого состояния σ

$$\phi_{(f_1,f_2)}(M[f_1](\sigma)) = M[f_2](\sigma)\big|_{\phi_{(f_1,f_2)}(\mathrm{Sobj}(f_1)\cup\mathrm{Obj}(S))},$$

Очевидно это отношение эквивалентности:

1) $S \approx_{op} S$ при тождественном отображении ϕ;
2) если $S \approx_{op} T$ при некотором ϕ, то $T \approx_{op} S$ при ϕ^{-1}, которое является всюду определенной биекцией, поскольку таково исходное ϕ;
3) если $S \approx_{op} T$ при некотором ϕ, а $T \approx_{op} U$ при некотором ψ, то $S \approx_{op} U$ при $\chi = \phi \circ \psi$.

Отношение семантического расширения в отличие от отношения семантической эквивалентности учитывает все объекты программы. Поскольку многие правила перевода вводят новые объекты, доказательство корректности системы правил перевода проводилось для этого отношения. При этом для каждого правила перевода проверялось выполнение условий из определения семантической эквивалентности. Поэтому мы имеем в качестве следствия сохранение эквивалентности. Пусть $R(S)$ обозначает программу, полученную в результате применения правила перевода R к программе S.

Теорема 1. Для любого правила перевода R и любой программы S на языке C-light программа $R(S)$ является семантическим расширением S.

Следствие. Для любого правила перевода R и любой программы S на языке C-light программа $R(S)$ семантически эквивалентна программе S.

Для обоснования корректности перевода помимо семантической эквивалентности исходной и новой программы необходимы также следующие важные свойства системы правил перевода:

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

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

Доказательства теорем и следствия можно найти в [3].

Заключение

Представленный в настоящей работе язык C-light лежит в основе проекта, цель которого — разработка метода и системы верификации С-программ. Достоинство языка C-light в том, что он составляет ориентированное на верификацию представительное подмножество языка С, позволяет работать с динамической памятью и обладает обозримой структурной операционной семантикой. Кроме того, язык C-light существенно расширяет язык Паскаль, для которого в 1991–1996 гг. была разработана система верификации программ [5].

Для верификации программ на языке C-light используется двухуровневый подход, позволяющий существенно упростить аксиоматическую семантику за счет перевода в язык C-kernel. Обоснование корректности этой системы правил не имеет аналогов в литературе и требует глубоких семантических исследований. Оказалось, что обычное отношение функциональной эквивалентности между исходной C-light программой и результирующей C-kernel программой неприменимо, и его пришлось ослабить посредством введения отношения семантического расширения, которое сохраняет свойства рефлексивности и транзитивности, однако не сохраняет свойства симметричности. В результате удалось провести полное доказательство корректности системы правил перевода, которое включает также доказательства завершимости процесса перевода и нормализации результирующего представления. Это позволило обосновать важное свойство, что частичная корректность исходной аннотированной C-light программы следует из частичной корректности результирующей C-kernel программы.

На основе системы правил перевода был создан прототип транслятора из языка C-light в язык C-kernel. На базе аксиоматической семантики языка C-kernel разработан прототип генератора условий корректности [6]. Предполагается разработать и реализовать экспериментальную систему верификации C-light программ, которая включает следующие основные части:

- транслятор из языка C-light в язык C-kernel;
- генератор условий корректности для языка C-kernel;
- доказатель условий корректности, включающий базы знаний о проблемных областях.

1. *Непомнящий В.А., Анореев И.С., Михайлов И.Н., Промский А.В.* На пути к верификации С программ. Язык C-light и его формальная семантика. // Программирование. - 2002. - № 6. - С. 1–13.
2. *Непомнящий В.А., Анореев И.С., Промский А.В.* На пути к верификации С программ. Аксиоматическая семантика языка C-kernel // Там же. - 2003. - № 6. - С. 5-15.
3. *Непомнящий В.А., Анореев И.С., Михайлов И.Н., Промский А.В.* Ориентированный на верификацию язык C-light. // Системная информатика: Сб. науч. тр. Института систем информатики имени А.П. Ершова, Вып. 9. - Новосибирск: Издательство СО РАН, 2004. - С. 51–134.
4. *Непомнящий В.А., Рякин О.М.* Прикладные методы верификации программ. - М.: Радио и связь, 1988.
5. *Непомнящий В.А., Сулимов А.А.* Проблемно-ориентированные базы знаний и их применение в системе верификации программ СПЕКТР // Изв. РАН. Сер. "Теория и системы управления". - 1997. - № 2. - С. 169–175.
6. *Промский А.В.* Генерация и метагенерация условий корректности в системе СПЕКТР-2. Новосибирск, 2003. – (Препр. ИСИ СО РАН; № 103).
7. *Programming* languages — C: ISO/IEC 9899:1999. - 566 p.
8. *Apt K.R.* Ten years of Hoare's logic: A survey - Part I // ACM Trans. Progr. Lang. and Systems. - 1981. - Vol. 3, № 4. - P. 431–483.
9. *Apt K.R., Olderog E.R.* Verification of sequential and concurrent programs. – Springer. - 1991.
10. *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. Lecture Notes in Computer Sci. - 1996. - Vol. 1125. - P. 56–60.
11. *Fradet P., Gaugne R., Le Metayer D.* Static detection of pointer errors: an axiomatization and a checking algorithm Proc. Europ. Symp. on Programming (ESOP96) // Lecture Notes in Computer Sci. - 1996. - Vol. 1058. - P. 125–140.
12. *Gurevich Y., Huggins J.K.* The semantics of the C programming language // Proc. of the Intern. Conf. on Computer Science Logic. Lecture Notes in Computer Sci. - 1993. - V. 702. - P. 274–309.
13. *Hoare C.A.R.* An axiomatic basis for computer programming // Communications ACM. - 1969. - Vol. 12. - № 1. - P. 576–580.
14. *Hoare C.A.R., Wirth N.* An axiomatic definition of the programming language Pascal // Acta Informatica. - 1973. - Vol. 2. - № 4. - P. 335–355.
15. *Huggins J.K., Shen W.* The static and dynamic semantics of C (extended abstract) // Local Proc. Int. Workshop on Abstract State Machines. ETH TIK-Rep. № 87, 2000. - P. 272–284.
16. *Nepomniashy V.A., Anureev I.S., Promsky A.V.* Verification-Oriented Language C-light and Its Structural Operational Semantics // Proc. of Conf. “Perspectives of System Informatics”, Springer-Verlaf, Berlin, LNCS. - 2003 - Vol. 2890. - P. 1-5.
17. *Norrish M.* Deterministic expressions in C // Proc. Europ. Symp. on Programming (ESOP99). Lecture Notes in Computer Sci. Vol. 1576. - 1999. - P. 147–161.
18. *Norrish M.* C formalized in HOL. PhD thesis. Computer Lab., Univ of Cambridge, 1998.
19. *Plotkin G.D.* A structure approach to operational semantics. (Tech. Rep./ DAIMI/Aarhus University; FN-19), 1981.