Модел. и анализ информ. систем. Т. **18**, № **4** (2011) 21–33

УДК 519.681

Атрибутные аннотации и их применение в дедуктивной верификации C-программ

Атучин М.М., Ануреев И.С.[1]

Институт систем информатики имени А.П. Ершова СО РАН

e-mail: anureev@iis.nsk.su, atuchinm@gmail.com

получена 9 октября 2011 года

Ключевые слова: дедуктивная верификация, атрибутная нормализация, атрибутные аннотации, аксиоматическая семантика, C-light, C-kernel

Предложен новый вид аннотаций, называемых атрибутными аннотациями, и методология их применения в дедуктивной верификации программ. Описана коллекция аннотирующих атрибутов для подмножества C-kernel языка C и на их основе представлены два варианта аксиоматической семантики языка C-kernel – семантика прямого прослеживания и смешанная семантика прямого прослеживания.

1. Введение

Использование упрощенных промежуточных языков — тренд в современных проектах, ориентированных на дедуктивную верификацию. Cminor (проект CompCert [1]), Why (проект Frama-C [2]), Boogie [3] (проекты Dafny [4] и VCC [5]) и Simpl (проект Verisoft [6]) — примеры промежуточных языков, используемых в проектах по верификации C-программ.

Трансляция аннотированных программ целевого языка в промежуточный язык элиминирует сложные конструкции целевого языка. В результате упрощается аксиоматическая семантика, которая разрабатывается только для промежуточного языка. В то же время трансляция расширяет область применения дедуктивного подхода к верификации на целевой язык.

Можно выделить два вида трансляции: трансформацию и нормализацию. Отличие между ними состоит в том, что в случае нормализации промежуточный язык является подмножеством целевого языка.

Преимуществом трансформации является то, что несколько различных целевых языков могут быть транслированы в промежуточный язык и аксиоматическую

[1]Работа частично поддержана грантом РФФИ 11-01-00028-а и интеграционной программой РАН 14/12.

семантику достаточно разработать только один раз для этого языка. Однако доказательство корректности таких трансляций обычно непростая задача. Все вышеприведенные примеры промежуточных языков попадают в эту категорию.

Наоборот, нормализация, как правило, выполняется для одного целевого языка. Исключением является трансляция различных диалектов или версий целевого языка с общим синтаксисом. Однако доказательство корректности нормализации обычно проще, чем в случае трансформации, так как целевой и промежуточный язык имеют одну и ту же операционную семантику.

Язык C-kernel [7], используемый в двухуровневом методе верификации C-программ [8, 9], попадает в эту категорию. Этот метод применяется к языку C-light — выразительному подмножеству языка ISO-C [13]. Чтобы верифицировать C-light-программы, мы сначала транслируем их в программы [10, 11] на языке C-kernel — ограниченном подмножестве языка C-light. Затем мы порождаем условия корректности, используя аксиоматическую семантику языка C-kernel [7]. Поскольку алгоритмы нормализации, применяемые для трансляции C-light-программ в C-kernel-программы, модифицируют исходный код транслируемой программы и исходные аннотации, в работе [11] доказывается корректность этих алгоритмов.

Хотя единая операционная семантика языков C-light и C-kernel упрощает доказательство корректности трансляции из C-light в C-kernel, сложность доказательства остается все еще достаточно высокой. Заметим, что под корректностью трансляции понимается не только функциональная эквивалентность исходной и преобразованной программ, но и сохранение истинности преобразованных аннотаций (например, инвариантов циклов и меток). Поэтому, чтобы иметь возможность выполнять дополнительные преобразования C-kernel-программ, нацеленные на дальнейшее упрощение генерации условий корректности, и обойти при этом проблему обоснования корректности этих преобразований, мы вводим специальный вид нормализации — атрибутную нормализацию, которая не изменяет исполняемые конструкции целевого языка и исходные программные аннотации, а только приписывает им атрибуты и присваивает этим атрибутам значения. Поскольку такая нормализация не влияет на исполнение программы и на истинность исходных аннотаций, она не требует доказательства корректности.

На основе этой идеи мы представляем новый двухшаговый метод упрощения генерации условий корректности — метод атрибутных аннотаций. На первом шаге программы целевого языка транслируются в программы промежуточного языка, используя только атрибутные нормализации. Пары – (аннотирующий) атрибут и его значение, приписываемые конструкциям и аннотациям исходной аннотированной программы, можно рассматривать как новый вид программных аннотаций, который мы называем атрибутными аннотациями. На втором шаге порождаются условия корректности для программ промежуточного языка. Атрибутные аннотации связывают дополнительную информацию с программными конструкциями и исходными программными аннотациями. Использование этой информации в правилах вывода аксиоматической семантики позволяет упростить как процесс генерации условий корректности, так и сами сгенерированные условия корректности.

Процесс генерации условий корректности упрощается за счет того, что используемая в правиле нелокальная информация об аннотированной программе становится

локально доступной. Например, правило для оператора *return e* требует нелокальной информации о типе выражения *e* и типе возвращаемого значения функции, в которой встречается этот оператор. Добавление соответствующих атрибутов к оператору *return* решает эту проблему.

Упрощение генерируемых условий корректности достигается за счет добавления специализированных правил вывода, использующих дополнительную информацию. Примером специализированного правила, используемого в смешанной аксиоматической семантике языка C-kernel [12], является правило для присваивания неразделяемой переменной. Значение переменной такого вида доступно при исполнении программы только через ее имя (а не через, например, указатель). Это позволяет применять к этой переменной специализированное правило, подобное правилу для языка Pascal, и получать более простые условия корректности по сравнению с применением общего правила для присваивания, используемого в обычной аксиоматической семантики языка C-kernel [7]. Дополнительная информация задается булевским атрибутом *nonshared*, который определяет, является ли переменная неразделяемой.

2. Двухуровневый метод верификации C-программ

Двухуровневый метод верификации C-программ [7, 8, 9] применяется к языку C-light — выразительному подмножеству языка ISO-C [13].

Тип void, целочисленные типы, вещественные типы и перечисления суть базовые типы языка C-light. Указатели, массивы, структуры и функции суть производные типы этого языка. Язык C-light покрывает все операторы языка C. Порядок вычисления выражений в языке C-light строго фиксирован. Аргументы операций и функций вычисляются справа налево, а инициализирующие списки выражений — слева направо. Особенностью C-light является то, что он имеет формальную операционную семантику [9].

Двухуровневый метод транслирует C-light-программу, которую требуется верифицировать в C-kernel-программу [10, 11] и применяет к последней правила аксиоматической семантики для языка C-kernel [7]. Все выражения в C-kernel находятся в нормальной форме. Число побочных эффектов в нормализованных выражениях сведено к минимуму и операции с контрольными точками (например, логические операции) исключены. Нормализованное выражение также не содержит условных операторов, оператора запятой, простых и составных присваиваний, операций инкремента и декремента. Списки деклараций допускаются только в декларациях функций, любая другая декларация определяет в точности один объект. Инициализаторы содержат только нормализованные выражения. Оператор выражения, оператор if с обязательной ветвью else и нормализованным условием, оператор while с нормализованным условием, оператор goto, оператор return с нормализованным выражением и блок суть операторы языка C-kernel.

Аксиоматическая семантика языка C-kernel [7] определяется как исчисление троек Хоара $\{P\}\, S\, \{Q\}$, где предусловие P и постусловие Q — формулы языка аннотаций, S — фрагмент программы на языке C-kernel. В качестве языка аннотаций

используется язык логики высших порядков. Аксиоматическая семантика базируется на абстрактной C-light-машине [8], которая определяет операционную семантику C-light и оперирует метапеременными посредством абстрактных функций.

Пусть $Locations$ — множество адресов и $CTypes$ — объединение всех типов языка C-light. Метапеременная $MD \in Locations \to CTypes$ определяет значения, хранимые в памяти. Метапеременная $Val \in CTypes$ хранит последнее значение, возвращенное функцией или операцией. Пусть $Names$ — множество всех имен, встречающихся в программе. Все программные переменные из $Names$ суть метапеременные C-light-машины.

Ниже мы определим только те абстрактные функции, которые используются в последующих разделах.

Пусть Nat — множество неотрицательных целых чисел. Функции $mem \in Names \to Location$ и $mb \in Names \times (Nat \cup Names) \to Locations$ описывают распределение памяти. Функция $mem(x)$ возвращает адрес идентификатора x. Функция $mb(e, id)$ возвращает адрес элемента $e[id]$ массива e или адрес поля $e.id$ структуры e. Функция $cast(e, \tau, \tau')$ преобразует значение e типа τ к типу τ'. Функция $type(e)$ возвращает тип выражения e. Функция $val(e, MD)$ вычисляет значение выражения e. Функция $addr(e, MD)$ вычисляет адрес выражения e. Их семантика определяется согласно стандарту [13].

Аксиоматическая семантика C-kernel использует те же самые метапеременные, что и C-light-машина, но изменяет семантику абстрактных функций. Функции mem, mb, $cast$ считаются неинтерпретированными. Они определяются на этапе доказательства условий корректности с помощью аксиом. Вместо функций $type$, val и $addr$ используются соответствующие аннотирующие атрибуты.

Обозначения. Пусть $P^{x_1/v_1,\ldots,x_n/v_n}$ обозначает одновременную замену всех вхождений метапеременных $x_1, \ldots, x_n$ в формуле P на $v_1, \ldots, v_n$. Мы также используем сокращенную векторную нотацию $P^{\overline{x}/\overline{v}}$. Пусть $upd(f, x, e)$ обозначает функцию f' такую, что $f'(y) = f(y)$ для любого $y \neq x$ и $f'(x) = e$. Мы также используем векторную нотацию $upd(f, (x_1, \ldots, x_n), e)$, которая означает, что значение функции f для аргументов $x_1, \ldots, x_n$ становится равным e.

3. Аннотирующие атрибуты для конструкций языка C-kernel

В этом разделе мы представим коллекцию аннотирующих атрибутов для конструкций языка C-kernel. Пусть $e.attr$ обозначает значение атрибута $attr$ для атрибутируемой конструкции e. Пусть $\underbrace{e}_{attr_1:val_1,\ldots,attr_n:val_n}$ обозначает тот факт, что конструкция e имеет атрибуты $attr_1, \ldots, attr_n$ и $val_1, \ldots, val_n$ суть значения этих атрибутов.

Выражения имеют три общих атрибута. Атрибут $type$ определяет тип выражения, атрибут val — значение выражения и атрибут $addr$ — адрес значения. Атрибут $addr$ определен только для l-значений (lvalues) [13]. Значение $e.type$ совпадает со значением $type(e)$ абстрактной функции $type$ C-light-машины. Значения атрибутов val и $addr$ — термы языка аннотаций. Значения этих термов совпадают со значениями

$val(e)$ и $addr(x)$ абстрактных функций val и $addr$ C-light-машины соответственно. Например, $(x+3).val = MD(mem(x)) + 3$ и $(x+3).addr = mem(x) + 3$.

Вызовы функций и переменные суть специальные случаи выражений, которые имеют дополнительные атрибуты.

Вызовы функций имеют три дополнительных атрибута pre, $post$ и $argtypes$. Если e — вызов функции f, то $e.pre$ — предусловие функции f, $e.post$ — постусловие функции f, $e.argtypes$ — список типов аргументов функции f.

Переменные имеют один дополнительный атрибут uid, который связывает с каждым вхождением переменной в программу уникальный идентификатор, устраняя таким образом коллизию имен, позволяющую разным объектам программы иметь одно и то же имя. Таким образом, разные объекты имеют разные значения атрибута uid.

Синонимы типов, определяемые декларациями типов, имеют атрибут uid, выполняющий ту же функцию, что и для переменных.

Типовые выражения имеют один атрибут $type$. Если e — типовое выражение, то $e.type$ — логический тип языка аннотаций, соответствующий выражению e. Например, если e имеет вид $(int(int*, char))$, то $e.type = pointer(int) \times char \to int$.

Блок имеет один атрибут $vars$, который определяет список уникальных идентификаторов переменных, объявленных в этом блоке на верхнем уровне. Например, если e имеет вид

$$\{int\ \underbrace{x}_{uid:x_1}; \{int\ \underbrace{x}_{uid:x_2}; \}\ int\ \underbrace{z}_{uid:z_1}; \underbrace{z}_{uid:z_1} = 2; \},$$

то $e.vars = (x_1, z_1)$.

Декларации переменных имеют один атрибут MD, значением которого является терм языка аннотаций, который описывает, как изменится метапеременная MD при инициализации переменных, объявленных в этой декларации. Например, если декларация e имеет вид

$$int\ \underbrace{x}_{uid:x_1}\ =\ 0;$$

то $e.MD = upd(MD, mem(x_1), 0)$.

Операторы $return$ имеют три атрибута $post$, $vars$ и $rettype$. Если e — оператор $return$, то $e.post$ — постусловие функции f, в которой встретился этот оператор, $e.rettype$ — тип значения, возвращаемого функцией f, $e.vars$ — список уникальных идентификаторов локальных переменных тела функции f, включая параметры этой функции.

Операторы goto имеют два атрибута inv и $vars$. Если e — оператор $goto$, то $e.inv$ — инвариант, связанный с e, $e.vars$ — список уникальных идентификаторов локальных переменных, которые элиминируются при передаче управления, инициированной этим оператором.

Оператор $while$ имеет один атрибут inv, определяющий инвариант этого оператора.

4. Аксиоматическая семантика языка C-kernel

В этом разделе мы рассмотрим модификацию аксиоматической семантики языка C-kernel [7], использующую аннотирующие атрибуты.

Присваивание. Пусть выражение e' не включает вызов функции. Правило для присваивания $e = e'$; имеет вид:

$$\frac{\{\exists MD'(P^{MD/MD'} \wedge MD = (upd(MD, c, cast(v', t', t)))^{MD/MD'})\}A\{Q\}}{\{P\} \underbrace{e}_{type:t,addr:c} = \underbrace{e'}_{type:t',val:v'} ; A\{Q\}}$$

Оно использует атрибуты *type* и *addr* для левой части присваивания e и атрибуты *type* и *val* для правой части присваивания e'.

Функциональный вызов. Правило для вызова функции f в случае, когда она возвращает значение, имеет вид:

$$\frac{\begin{array}{l}\{\exists MD''\,\exists MD'\,\exists Val' \\ (\forall a((P \Rightarrow P'^{arg_1/cast(v_1,t_1,t'_1)\ldots,arg_n/cast(v_n,t_n,t'_n)})^{MD/MD',Val/Val'} \Rightarrow \\ \quad (Q'^{arg_1/(cast(v_1,t_1,t'_1))^{MD/MD',Val/Val'},\ldots,arg_n/(cast(v_n,t_n,t'_n))^{MD/MD',Val/Val'}})^{MD/MD''})\wedge \\ MD = upd(MD'', c, cast(Val, t', t))\}A\{Q\}\end{array}}{\{P\} \underbrace{e}_{type:t,addr:c} = \underbrace{f(\underbrace{e_1}_{val:v_1,type:t_1}, \ldots, \underbrace{e_n}_{val:v_n,type:t_n})}_{type:t',argtypes:(t'_1,\ldots,t'_n),pre:P',post:Q'} ; A\{Q\}}$$

Оно использует атрибуты *type* и *addr* для левой части присваивания, атрибуты *type*, *argtypes*, *pre* и *post* для вызова функции f и атрибуты *type* и *val* для аргументов функции f. Здесь a — параметр тройки Хоара для функции f, который может входить в предусловие P' и постусловие Q' этой функции.

Правило для вызова функции f в случае, когда она не возвращает значение, имеет вид:

$$\frac{\begin{array}{l}\{\exists MD' \\ (\forall a((P \Rightarrow P'^{arg_1/cast(v_1,t_1,t'_1)\ldots,arg_n/cast(v_n,t_n,t'_n)})^{MD/MD'} \Rightarrow \\ \quad Q'^{arg_1/(cast(v_1,t_1,t'_1))^{MD/MD'},\ldots,arg_n/(cast(v_n,t_n,t'_n))^{MD/MD'}})\}A\{Q\}\end{array}}{\{P\}\, \underbrace{f(\underbrace{e_1}_{val:v_1,type:t_1}, \ldots, \underbrace{e_n}_{val:v_n,type:t_n})}_{argtypes:(t'_1,\ldots,t'_n),pre:P',post:Q'} ; A\{Q\}}$$

Оно использует *argtypes*, *pre* и *post* для вызова функции f и атрибуты *type* и *val* для аргументов функции f.

Декларация переменной. Правило для декларации переменной e имеет вид:

$$\frac{\{\exists MD'(P^{MD/MD'} \wedge MD = v^{MD/MD'})\}A\{Q\}}{\{P\}\underbrace{e}_{MD:v};A\{Q\}}$$

Оно использует атрибут MD, который специфицирует инициализацию переменной, объявляемой в декларации e.

Помеченные операторы. Правило для помеченного оператора имеет вид:

$$\frac{\{P\}\,S\,A\,\{Q\}}{\{P\}\,L:S\,A\,\{Q\}}$$

Оно является единственным правилом, которое не использует атрибуты.

Блок. Правило для блока имеет вид:

$$\frac{\{P\}\,S\,DelVars(x_1,\ldots,x_n)\,A\,\{Q\}}{\{P\}\quad\underbrace{\{S\}}_{vars:(x_1,\ldots,x_n)}\quad A\,\{Q\}}$$

Оно использует атрибут $vars$ для блока. Дополнительная конструкция $DelVars(x_1,\ldots,x_n)$ делает переменные $x_1,\ldots,x_n$ неопределенными:

$$\frac{\{\exists MD'(P^{MD/MD'} \wedge MD = upd(MD',(mem(x_1),\ldots,mem(x_n)),\omega))\}\,A\,\{Q\}}{\{P\}\,DelVars(x_1,\ldots,x_n)\,A\,\{Q\}}$$

Здесь ω обозначает неопределенное значение.

Оператор if. Правило для оператора if имеет вид:

$$\frac{\{P \wedge cast(v,\,t,\,int) \neq 0\}\,S_1\,A\,\{Q\}\quad\{P \wedge cast(v,\,t,\,int) = 0\}\,S_2\,A\,\{Q\}}{\{P\}\,if(\underbrace{\quad e\quad}_{val:v,type:t})\,S_1\,else\,S_2\,A\,\{Q\}}$$

Оно использует атрибуты val и $type$ для управляющего выражения e условного оператора if.

Оператор $while$. Правило для оператора $while$ имеет вид:

$$\frac{P \Rightarrow I\quad\{INV \wedge cast(v,t,int) \neq 0\}\,S\,\{I\}\quad\{INV \wedge cast(v,t,int) = 0\}\,A\,\{Q\}}{\underbrace{\{P\}\;while(\underbrace{\quad e\quad}_{val:v,type:t})\;S\;A\,\{Q\}}_{inv:I}}$$

Оно использует атрибут inv для цикла $while$ и атрибуты val и $type$ для условия этого цикла.

Операторы передачи выполнения. Операторы *goto* и *return* суть операторы передачи выполнения языка C-kernel.

Правило для оператора *goto* имеет вид:

$$\frac{\{P\}DelVars(x_1,\ldots,x_n)\{I\}}{\{P\}\quad \underbrace{goto\,L;}_{inv:I,vars:(x_1,\ldots,x_n)}\quad A\,\{Q\}}$$

Оно использует атрибуты *inv* и *vars* для оператора *goto*.

Семантика оператора *return* определяется двумя правилами в зависимости от того, имеется ли возвращаемое значение.

Правило для оператора *return* с аргументом завершает текущий вызов функции, передает управление точке, в которой постусловие функции истинно, и возвращает значение в метапеременной Val:

$$\frac{\{\exists Val'(P^{Val/Val'} \wedge Val' = cast(v,t,t'))\}\; DelVars(x_1,\ldots,x_n)\,\{Q'\}}{\{P\}\quad \underbrace{return\quad \underbrace{e}_{val:v,type:t}\;;}_{post:Q',vars:(x_1,\ldots,x_n),rettype:t'}\quad A\,\{Q\}}$$

Оно использует атрибуты *post*, *vars* и *rettype* для оператора *return* и атрибуты *val* и *type* для выражения этого оператора.

Правило для оператора *return* без аргумента завершает текущий вызов функции и передает управление в точку, в которой постусловие функции истинно:

$$\frac{\{P\}\; DelVars(x_1,\ldots,x_n)\,\{Q'\}}{\{P\}\quad \underbrace{return;}_{post:Q',vars:(x_1,\ldots,x_n)}\quad A\,\{Q\}}$$

Оно использует атрибуты *post* и *vars* для оператора *return*.

5. Смешанная аксиоматическая семантика языка C-kernel

Понятие "смешанная аксиоматическая семантика" означает, что могут быть несколько правил вывода для одной и той же программной конструкции, которые применяются в зависимости от контекста. В этом разделе мы рассмотрим специальный случай смешанной аксиоматической семантики языка C-kernel [12], использующий булевский атрибут *nonshared*. Этот атрибут определяет, является ли переменная неразделяемой. Переменная неразделяема, если ее значение доступно только через ее имя. Неразделяемыми могут быть как простые переменные, так и структуры и массивы. Для последних это означает, что их элементы доступны только через имя соответствующего массива или структуры. Это свойство используется в смешанной аксиоматической семантике языка C-kernel для определения специализированных правил для конструкций этого языка, упрощающих порождаемые условия корректности.

Мы рассмотрим специализированные правила для присваивания простой неразделяемой переменной, элементу неразделяемого массива и полю структуры. Они используют упрощенную модель памяти, подобную модели, применяемой в классической логике Хоара для языка Паскаль, и позволяют за счет этого упростить генерируемые условия корректности.

Правило для присваивания $x = e$, где x — простая неразделяемая переменная, имеет вид:

$$\frac{\{\exists z\ P^{y/z}\ \wedge (y = cast(v^{y/z}, t, s))\}\ A\,\{Q\}}{\{P\}\quad \underbrace{x}_{nonshared{:}true,uid{:}y,type{:}s} \quad = \quad \underbrace{e}_{val{:}v,type{:}t}\ ;\ A\,\{Q\}}$$

Дополнительно к атрибуту *nonshared* оно использует атрибуты *uid* и *type* для переменной x, которой присваивается значение, и атрибуты *val* и *type* для присваиваемого выражения e.

Правило для присваивания $x[e] = e'$, где x — неразделяемый массив, имеет вид:

$$\frac{\{\exists z\ P^{y/z}\ \wedge (y = (upd(y, cast(v, t, int), cast(v', t', s)))^{y/z})\}\ A\,\{Q\}}{\{P\}\quad \underbrace{\underbrace{x}_{nonshared{:}true,uid{:}y}\ [\ \underbrace{e}_{val{:}v,type{:}t}\ ]}_{type{:}s} = \quad \underbrace{e'}_{val{:}v',type{:}t'}\ ;\ A\,\{Q\}}$$

Дополнительно к атрибуту *nonshared* оно использует атрибут *type* для левой части присваивания, атрибут *uid* для массива x, элементу которого присваивается значение, и атрибуты *val* и *type* для индекса e и для присваиваемого выражения e'.

Правило для присваивания $x.m = e$, где x — неразделяемая структура, имеет вид:

$$\frac{\{\exists z\ P^{y/z}\ \wedge (y = (upd(y, m, cast(v, t, s)))^{y/z})\}\ A\,\{Q\}}{\{P\}\quad \underbrace{\underbrace{x}_{nonshared{:}true,uid{:}y}\ .m}_{type{:}s} = \quad \underbrace{e}_{val{:}v,type{:}t}\ ;\ A\,\{Q\}}$$

Дополнительно к атрибуту *nonshared*, оно использует атрибут *type* для левой части присваивания, атрибут *uid* для структуры x, полю которого присваивается значение, и атрибуты *val* и *type* для присваиваемого выражения e.

6. Пример

Использование атрибутной информации в правилах аксиоматической семантики иногда позволяет достичь существенного выигрыша при генерации условий корректности. В этом разделе мы рассмотрим такой предельный случай на примере генерации условий корректности для C-kernel-программы, последовательно присваивающей значения переменным x_1, ..., x_n:

$$\begin{array}{l}\{P(x_1, \ldots, x_n)\}\\ x_1 = e_1(x_1, \ldots, x_n);\\ \ldots\\ x_n = e_n(x_1, \ldots, x_n);\\ \{Q(x_1, \ldots, x_n)\}\end{array}$$

Чтобы упростить оценку результирующего присваивания, мы используем семантику обратного прослеживания для оператора присваивания (вместо прямого прослеживания, описанного в предыдущих разделах) и считаем, что выражения e_1, ..., e_n и формулы P и Q содержат ровно одно вхождение каждой из переменных x_1, ..., x_n. Результирующее условие корректности VC (оно единственное) будем оценивать по числу вхождений переменных, которое обозначим $ocnum(VC)$.

В случае применения обычной неатрибутной аксиоматической семантики, мы не обладаем никакой дополнительной информацией о переменных x_1, ..., x_n и поэтому используем общую модель памяти для C-light-программ [11]. В этом случае программу можно переписать в следующий набор инструкций C-light-машины:

$$\begin{array}{l}\{P(MD(mem(x_1)),\ldots,MD(mem(x_n)))\}\\ MD := upd(MD, mem(x_1), e_1(MD(mem(x_1)),\ldots,MD(mem(x_n))));\\ \ldots\\ MD := upd(MD, mem(x_n), e_n(MD(mem(x_1)),\ldots,MD(mem(x_n))));\\ \{Q(MD(mem(x_1)),\ldots,MD(mem(x_n)))\}\end{array}$$

для которых уже можно применять классическое паскалевское правило обратного прослеживания логики Хоара $\frac{\{P\}\,A\,\{Q(x\leftarrow e)\}}{\{P\}\,A\ x:=e\,\{Q\}}$.

Формула VC, полученная в результате применении этого правила в комбинации с правилом $\frac{P\Rightarrow Q}{\{P\}\,\{Q\}}$ к набору инструкций, имеет вид $P \Rightarrow Q'$, а ее сложность $ocnum(VC) = n + n * (n+1)^n$. Заметим, что в качестве переменной, вхождения которой учитываются, выступает метапеременная MD абстрактной C-light-машины, а переменные x_1, ..., x_n считаются константами из области определения функции MD.

Рассмотрим теперь случай атрибутных правил. Предположим, что в результате статического анализа мы установили, что все переменные x_1, ..., x_n являются неразделяемыми. В этом случае мы можем использовать упрощенную модель памяти C-light-машины для этих переменных и применять к ним классическое паскалевское правило обратного прослеживания $\frac{\{P\}\,A\,\{Q(x\leftarrow e)\}}{\{P\}\,A\ \underbrace{x}_{nonshared:true}\ =e;\,\{Q\}}$, добавив к нему атрибут *nonshared*.

Формула VC, полученная в результате применения этого правила к C-kernel-программе, имеет вид $P \Rightarrow Q'$, а ее сложность $ocnum(VC) \leq n + n * 2^n$.

Таким образом, использование атрибутной аксиоматической семантики обеспечивает нам значительный выигрыш с ростом n.

Заметим, что хотя мы и рассмотрели предельный случай, когда все переменные C-kernel-программы являются неразделяемыми, в реальных C-программах такие переменные (например, локальные переменные, счетчики циклов) встречаются достаточно часто.

7. Заключение

Результаты, представленные в этой работе, включают: концептуализацию знаний о применении атрибутных техник в дедуктивной верификации программ (вводятся

понятия атрибутной нормализации и атрибутной аннотации), новый метод упрощения генерации условий корректности, основанный на атрибутных аннотациях, коллекцию атрибутных аннотаций для языка C-kernel и два варианта аксиоматической семантики для этого языка — семантику прямого прослеживания [7] и смешанную семантику прямого прослеживания [12], модифицированных с учетом атрибутных аннотаций.

Метод атрибутных аннотаций может использоваться, чтобы:

- комбинировать статический анализ с дедуктивной верификацией. Результаты статического анализа описываются атрибутными аннотациями, которые затем используются в правилах аксиоматической семантики. Например, с помощью статического анализа можно собирать информацию о неразделяемых переменных. В частности, это позволяет использовать информацию, получаемую обычно при компиляции программы, в правилах аксиоматической семантики. Например, константное выражение может быть аннотировано атрибутом, который определяет значение этого выражения;

- упростить трансляцию из целевого языка в промежуточный и доказательство корректности этой трансляции. Это достигается заменой трансформаций конструкций целевого языка их атрибутированием. Например, расстановка пропущенных спецификаторов класса памяти в декларациях переменных, которая выполняется при трансляции C-light-программ в C-kernel-программы [10], может быть заменена добавлением соответствующих атрибутов к этим декларациям;

- комбинировать автоматическое доказательство с порождением условий корректности. Такой подход использовался в проблемно-ориентированной версии системы СПЕКТР для языка Паскаль [14, 15]. В этом случае атрибуты специфицируют, какие формулы, появляющиеся в процессе генерации условий корректности, доказываются на месте, и как результаты их доказательства встраиваются в этот процесс;

- управлять процессом генерации условий корректности. В частности, атрибуты могут определять стратегии вывода условий корректности (порядок применения правил аксиоматической семантики) и специфицировать контекст этого процесса. Например, контекст, используемый в правилах аксиоматической семантики для языка C#-kernel [16], специфицирует ситуацию, в которой вывод условий корррректности проводится в режиме распространения исключений (exception propagation) до тех пор, пока не встретится их обработчик. Эта ситуация возникает, когда оператор *throw* или среда исполнения инициируют исключение. В этом случае атрибуты определяют значение и тип инициируемого исключения;

- описывать обратные связи, позволяющие специфицировать места в исходной программе, которые соответствуют определенным порожденным условиям корректности и контрпримерам, в случае ложных условий корректности.

Одним из модулей системы СПЕКТР-1, реализующей двухуровневый метод верификации С-программ [9, 8], является модуль трансляции C-light-программ в C-kernel-программы. Мы планируем расширить этот модуль атрибутными аннотациями и интегрировать в новую мультиязыковую версию системы СПЕКТР [18]. В качестве языка реализации модуля планируется использовать встроенный в эту версию предметно-ориентированный язык выполнимых спецификаций Atoment [17], предназначенный для разработки подобных модулей.

Список литературы

1. Leroy X. Formal verification of a realistic compiler // Communications of the ACM. 2009. 52(7). P. 107–115.

2. Filliâtre J. C., Marché C. Multi-prover verification of C programs // Proc. ICFEM 2004. 2004. LNCS 3308. P. 15–29.

3. Leino K.R.M., Rümmer P. A polymorphic intermediate verification language: Design and logical encoding // TACAS 2010. 2010. LNCS 6015. P. 312–327.

4. Leino K. R. M. Dafny: an automatic program verifier for functional correctness // LPAR-16. 2010. LNCS 6355. P. 348–370.

5. Cohen E., Dahlweid M, Hillebrand M., et al. VCC: A Practical System for Veirying Concurrent C // TPHOLs 2009. 2009. LNCS 5674. P. 23–42.

6. Alkassar E., Hillebrand M.A., Leinenbach D., Schirmer N.W., Starostin A. The Verisoft Approach to Systems Verification // VSTTE 2008. 2008. LNCS 5295. P. 209–224.

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

8. Nepomniaschy, V. A., Anureev, I. S., Promsky, A. V. Verification-oriented language C-light and its structural operational semantics // Proc. of Conf. PSI-2003. 2003. LNCS 2890. P. 1–5.

9. 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.

10. Непомнящий В.А., Анореев И.С., Промский А.В. На пути к верификации С-программ. Язык C-light и его трансформационная семантика // Проблемы программирования. 2006. № 2-3. С. 359–368.

11. Непомнящий В.А., Анореев И.С., Михайлов И.Н., Промский А.В. Ориентированный на верификацию язык C-light // Формальные методы и модели информатики: Сборник научных трудов. Серия "Системная информатика". Новосибирск: Издательство СО РАН. 2004. № 9. С. 51–134.

12. Ануреев И.С., Марьясов И.В., Непомнящий В.А. Верификация С-программ на основе смешанной аксиоматической семантики // Моделирование и анализ информационных систем. 2010. Том 17, № 3. С. 5–28.

13. ISO/IEC 9899: Programming languages – C. (1999)

14. Nepomniaschy V.A., Sulimov A.A. Promlem-oriented means of program specification and verification in project SPECTRUM // DISCo 1993. 1993. LNCS 722. P. 374–378.

15. Nepomniaschy V.A., Sulimov A.A. Problem-oriented verification system and its application to linear algebra programs // Theoretical Computer Science. 1993. 119. P. 173–185.

16. Непомнящий В.А., Ануреев И.С., Промский А.В., Дубрановский И.В. На пути к верификации C# программ: трехуровневый подход // Программирование. 2006. № 4. С. 4–20.

17. Anureev I.S. Introduction to the Atoment language // Joint NCC&IIS Bulletin, Series Computer Science. 2010. Vol. 30. P. 1–16.

18. Непомнящий В.А., Ануреев И.С., Атучин М.М., Марьясов И.В., Петров А.А., Промский А.В. Система анализа и верификации С-программ СПЕКТР-2 // Моделирование и анализ информационных систем. 2010. Том 17, № 4. С. 88–100.

Attribute Annotations and Their Use in C Program Deductive Verification

Atuchin M.M., Anureev I.S.

Keywords: deductive verification, attribute normalization, attribute annotations, axiomatic semantics, C-light, C-kernel

In this paper a new kind of annotations, called attribute annotations, and the methodology for their application in a deductive program verification are proposed. A collection of annotating attributes for the subset C-kernel of the C language is described, and on their base two versions of axiomatic semantics of C-kernel – forward semantics and mixed forward semantics – are presented.

Сведения об авторах:
Атучин Михаил Михайлович,
Институт систем информатики имени А.П. Ершова СО РАН, аспирант;
Ануреев Игорь Сергеевич,
Институт систем информатики имени А.П. Ершова СО РАН,
старший научный сотрудник