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

И.В Каблуков, В.И. Шелехов

КОНТРОЛЬ ДИНАМИЧЕСКОЙ СЕМАНТИКИ ПРЕДИКАТНОЙ ПРОГРАММЫ

Препринт
162

Новосибирск 2012

Описывается реализация контроля динамической семантики предикатных программ в системе предикатного программирования методом доказательства условий семантической корректности в системе автоматического доказательства PVS. Условия семантической корректности программы транслируются в систему автоматического доказательства PVS.

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

I.V. Kablukov, V.I. Shelekhov

CHECKING THE DYNAMIC SEMANTICS CONDITIONS OF PREDICATE PROGRAMS

Preprint
162

Novosibirsk 2012

In the predicate programming system, the development of checking the dynamic semantics conditions of predicate programs by means of automatic proof in the PVS system is described.

1. ВВЕДЕНИЕ[1]

Семантический анализ в трансляторе с языка программирования проводится в соответствии с правилами языка, определяющими условия семантической корректности конструкций программы. Условия корректности определяются на значениях атрибутов конструкций. Транслятор может проверить лишь часть условий – они относятся к статической семантике. Все остальные условия, которые транслятор не может проверить сам, относятся к динамической семантике.

Существуют различные подходы к реализации контроля динамической семантики. Типичным решением является вставка в код оператора assert(<предикат>) при трансляции в отладочном режиме. Для поиска ошибок (не только динамической семантики) используются статические анализаторы, базирующиеся на потоковом анализе программы. Частным случаем статических анализаторов является абстрактная интерпретация – результатами псевдоисполнения программы являются не значения переменных, а формулы, содержащие эти переменные. В данном проекте реализуется наиболее сильный метод, ранее не применявшийся для функциональных и императивных языков: условия семантической корректности программы транслируются в систему автоматического доказательства и там доказываются. Задача проекта – реализовать данный метод в системе предикатного программирования [1].

Разработаны формальные условия семантической корректности для конструкций языка P [2] (разд. 3). В разд. 2 описывается задача сравнения типов, возникающая при семантическом анализе большинства конструкций. В разд. 4 содержится обзор методов контроля динамической семантики в системе автоматического доказательства PVS, реализуемый (как и в нашем проекте) через доказательство условий корректности. Набор условий корректности оформляется в виде набора теорий на языке внутреннего представления предикатного программирования (разд. 5). Далее формулы корректности транслируются на язык спецификаций системы PVS [3]. Отображение из языка внутреннего представления в язык системы PVS описано в разд. 6. Для проведения доказательства составляются стратегии и методы автоматического доказательства формул в системе PVS в интеграции

[1] Работа выполнена при поддержке РФФИ, грант 12-01-00686.

с SMT-решателем Yices [8] (разд. 7). В заключении описываются результаты проекта.

2. ЗАДАЧА СРАВНЕНИЯ ТИПОВ

Программа на языке P – это список определений предикатов следующего вида:

A(x: y) **pre** P(x) { S(x: y) } **post** Q(x, y) ,

где x, y – входные и выходные параметры, S(x: y) – *оператор*, а P(x) и Q(x, y) – предусловие и постусловие; предусловие должно быть истинным перед исполнением S(x: y), а постусловие – после исполнения.

Семантика – это система правил, определяющих условия на значениях атрибутов конструкций языка. Например, для оператора присваивания x := E тип выражения E должен быть совместим с типом переменной x. Совместимость типов требует вложенности одного типа в другой. Таким образом, должно выполняться условие $T_E \subseteq T_x$, где T_x– тип x, T_E – тип E.

Сравнение двух типов является типичной задачей семантического анализа программы.

Описание типа определяется конструкцией:

type <имя типа>(<параметры типа>) = <типовой терм>.

Всякий тип может быть представлен типовым термом. Имеются следующие виды типовых термов:

bool, int, nat, real, char – булевые, числовые, символьные;

array(Tval, T_1 , … , T_n) – массив, Tval – тип элементов массива, T_i – типы индексов (размерностей массива);

set(T) – множество подмножеств типа T;

subtype(T v: P(v, p)) – подтип типа T, v – переменная подтипа, P – предикат (условие принадлежности к этому подтипу), p – параметр типа;

struct(T_1 a_{11}, … , a_{1n}, … , T_k a_{k1}, … , a_{km}) – структура, значением является совокупность значений полей a_{ij}, T_i – типы полей.

union($constr_1$(T_1 a_{11}, … , a_{n1}, … , T_k a_{1k}, … , a_{nk}); … $constr_n$(…)) – объединение, значением типа объединения является один из конструкторов с набором своих полей, a_{ij} – поля конструктора, T_i – типы полей.

predicate(T_{a1}, … , T_{an} **:** T_{r1}, … , T_{rn}) – предикатный тип, T_{ai} – типы входных параметров, T_{ri} – типы результирующих параметров;

Сравнение типов реализуется сравнением типовых термов, представляемых деревьями: вершинами являются конструкции, сыновьями вершин

– их подконструкции. Деревья должны совпадать, за исключением, возможно, листьев. Для листьев далее решается задача включения типов.

Структурное совпадение типовых термов проверяется транслятором. Для листьев включение типов определяется следующим образом: допустим типы T1 и T2 – подтипы типа T с предикатами P1 и P2. Нужно определить, что T1 – подмножество T2: $T_1(x_1, \dots, x_n) \subseteq T_2(y_1, \dots, y_n)$ (наборы x и y – параметры типов). Данное условие выражается в виде утверждения $P1(x_1, \dots, x_n) \Rightarrow P2(y_1, \dots, y_n)$. Данное утверждение является *условием семантической корректности*, ниже будут перечислены ситуации, в которых они генерируются.

3. УСЛОВИЯ КОРРЕКТНОСТИ ДИНАМИЧЕСКОЙ СЕМАНТИКИ

Условия семантической корректности оформляются в виде лемм со следующей структурой:

<имя леммы>: lemma <префикс> => <утверждение>

<префикс> состоит из конъюнкции предусловия и условий в условных операторах, обрамляющих текущую конструкцию.

P & C_1 & … & C_n; C_i – после if, ***not*** C_i – после else.

Если <префикс> пустой, телом леммы является <утверждение>; представляющее собственно условие корректности для каждой конкретной ситуации.

3.1. Совместимость типов

Совместимость типов – это вложенность одного типа в другой (типы рассматриваются как множества). Ниже понятие совместимости раскрывается для различных классов типов.

A. Примитивные типы:

nat совместим с **int**, **int** совместим с **real**

bool, **char** совместимы только сами с собой

B. Предикатные типы:

predicate (. . .) **pre** P_1 **post** Q_1

predicate (. . .) **pre** P_2 **post** Q_2

Для совместимости первого предикатного типа со вторым необходимо, чтобы число входных и выходных параметров совпадали, типы входных параметров были совместимы, типы выходных параметров совпадали и были истинны формулы $Q_2 \Rightarrow Q_1$ & $P_1 \Rightarrow P_2$.

C. Подтипы:

Подтип совместим с простым типом, если базовый тип подтипа совместим с данным простым типом.

Для подтипа с подтипом:

type T = **subtype**(A a: Texpr) // Texpr – логическое выражение, зависящее от переменной a

type S = **subtype**(B b: Sexpr)

Для совместимости типов T и S необходимо, чтобы тип B был совместим с типом A, и должна быть истинна формула Sexpr => Texpr.

D. Перечисление:

Элементы должны совпадать

E. Структурные типы:

Структура: количество и имена полей должны совпадать, типы соответствующих полей совместимы.

Объединение: количество полей и конструкторы соответствующих полей должны совпадать.

Множество: базовые типы множеств должны быть совместимы.

Массив: типы элементов должны быть совместимы, типы измерений массива должны совпадать.

В следующих разделах описывается набор ситуаций в языковых конструкциях и генерируемые условия семантической корректности.

3.2. Соответствие типов аргументов вызова и типов аргументов в определении предиката

Конструкции:

A(. . . y . . . : . . .) **pre** … { … } **post** … – определение предиката

A(. . . x . . . : . . .) – вызов предиката в некоторой части программы

Атрибуты: T_x – тип выражения (аргумента вызова) x, T_y – тип аргумента y.

Правило: T_x совместим с T_y.

Условие корректности:

Если T_x не равен T_y, то проверяется совместимость T_x с T_y.

Имя леммы: Call*номер*

3.3. Совпадение типов результатов вызова и типов результатов в определении предиката

Конструкции:

P(. . . : . . . A . . .) **pre** … { … } **post** … определение предиката

P(. . . : . . . B . . .) вызов предиката

Атрибуты: A и B – результаты предикатов, T_A и T_B типы переменных A и B.

Правило: T_A совпадает с T_B.

Условие корректности:

Проверяется равенство типов T_A и T_B.

Например, в случае массивов проверяется равенство типов элементов массивов и равенство размерностей.

Имя леммы: CallResult*номер*

3.4. Присваивание

Конструкция:

x = E;

Атрибуты: T_x – тип переменной x, T_E – тип выражения E.

Правило: тип T_x совместим с типом T_E. В частности, проверяется что T_x является подтипом T_E.

Условие корректности:

Если T_x не равен T_E, то проверить совместимость T_x с T_E.

Имя леммы: Assign*номер*

3.5. Оператор выбора

Конструкция:

switch(x) {

case alt_1 : …

case alt_2 : …

…

case alt_n : …

default : …

}

Атрибуты: выражения или диапазоны alt_1, alt_2, . . . , alt_n (если alt_i – диапазон, то обозначим $min(alt_i)$ – его нижнюю границу, $max(alt_i)$ – верхнюю), тип выражения x (конечный в случае отсутствия default'a).

Правило: значения альтернатив попарно различны (кроме проверок, которые делаются статически). Если нет альтернативы по умолчанию (default), то проверить, что объединение альтернатив покрывает тип выражения x.

Условие корректности:

Попарное непересечение альтернатив:

Если alt_i, alt_j – выражения: $alt_i != alt_j \; \forall \; i != j$.

Если alt_i – выражение, alt_j – диапазон: $(alt_i < min(alt_j)) \; \& \; (alt_i > max(alt_j)) \; \forall \; i ! = j$. Если alt_i, alt_j – диапазоны: $\forall \; i != j \; (min(alt_i) > max(alt_j)) \vee (max(alt_i) < min(alt_j))$.

При отсутствии default:

Пусть a_i есть $(x = alt_i)$ для выражения,

$((x >= min(alt_i)) \; \& \; (x <= max(alt_i)))$ для диапазона.

Проверяемое условие: $a_1 \vee a_2 \vee . . . \vee a_n$.

Имя леммы: Switch*номер*

3.6. Условный оператор

Конструкция:

if (A) { … }

else { … }

Атрибуты: логическое выражение A и логическое условие D, составляемое из конъюнкции предусловия и условий в условных операторах, обрамляющих текущую конструкцию.

Это дополнительное правило, не относящееся к динамической семантике. Если оно нарушается, то, возможно, ранее в программе была допущена ошибка.

Правило: Ветви **if** и **else** должны быть достижимы.

Условие корректности:

$\exists x \; D \; \& \; A$

$\exists x \; D \; \&$ **not** A

x – набор переменных, находящихся в D и A.

Имя леммы: If*номер*

3.7. Проверка выхода индекса за границы массива

Объявление типа массива представляется в виде:

type <имя> = **array**(<тип элементов>, <список типов измерений>)

Конструкции:

type M = **array** (T, D) – объявление типа массива, D есть список типов D_1, … , D_n; D_i – конечный тип i-го измерения массива. Как правило, он представляется подтипом, пусть P_i – предикат данного подтипа.

M m ; – описание массива

m[…, i_k, …] – вхождение элемента массива

Атрибуты: D_k – тип k-того измерения массива m, предикат P_k и номер i_k.

Правило: i_k принадлежит типу D_k, т.е. для i_k истинен предикат P_k.

Условие корректности:

$P_k(i_k)$ должно быть истинно.

Например, если D_k – диапазон $0..n_k$, то условие корректности:

$(0<= i_k) \& (i_k <= n_k)$

Имя леммы: Array*номер*

3.8. Неделение на ноль

Конструкция:

y/x

Правило: x не равно нулю.

Условие корректности:

x ! = 0

Имя леммы: Divide*номер*

3.9. Определение массива по частям

Определение массива представляется в виде:

for (<индексы определения>) { <определение частей массива> }

<определение частей массива>::=

(<определение части массива>)+ [**default** : <определение элемента>]

<определение части массива> ::=

case <индекс> : <определение элемента>

Пример:

for (**var** i) { **case** 1..10 : i+1 **case** 11..19 : 2*i **case** 20 : -1 **default** : 0 }

Конструкция:

M = array(T, D) M m;

m = for(var j) { case A_1 : . . . case A_2: … … case A_n: . . . default: … }

Атрибуты: список типов измерений массива D, диапазоны или выражения A_i.

Правило: A_i попарно не пересекаются и не выходят за границы D. Если нет **default**, то объединение A_i должно быть равно D.

Условие корректности:

Попарное непересечение альтернатив:

Если A_i, A_j – выражения: $A_i != A_j \ \forall\ i != j$.

Если A_i – выражение, A_j – диапазон:

$(A_i < \min(A_j)) \ \&\ (A_i > \max(A_j)) \ \ \forall i!=j$.

Если A_i, A_j – диапазоны:

$(\min(A_i) > \max(A_j)) \vee (\max(A_i) < \min(A_j)) \ \ \forall i!=j$.

Невыход за границы массива:

Если D – диапазон:

Если A_i – диапазон: $(\min(A_i) >= \min(D)) \ \&\ (\max(A_i) <= \max(D))$.

Если A_i – выражение: $(A_i >= \min(D)) \ \&\ (A_i <= \max(D))$.

При отсутствии **default**:

$A1 \cup A2 \cup \ldots \cup An = D$. A_i – для диапазона, $\{A_i\}$ – для выражения.

Имя леммы: ArrayPart*номер*

3.10. Модификация массива

Определение модификации массива представляется в виде:

<имя массива> **with** [<модификация массива>]

<модификация массива> ::= <индекс изменяемого элемента массива>:
<элемент массива> (, <модификация массива>)+

Пример:

type Vec(**nat** n) = **array** (**real**, 1..n);

perm(**nat** n, Vec(n) b, **nat** m, k, i, j : Vec(n) b')

{ b' = b **with** [k: b [m], m: b [k]]}; //обмен элементов с индексами k и m.

Атрибуты: массивы с диапазоном 1..n, номера перестановок m, k (многомерные в общем случае).

Правило: номера перестановок не выходят за границы массива и попарно не равны между собой.

Условия корректности:

(k >= 1) & (k <= n) & (m >= 1) & (m <= n)

k != m

Имя леммы: ArrayMod*номер*

3.11. Объединение массивов

Конструкция:

array(T, D_1) M_1, **array**(T, D_2) M_2, . . . **array**(T, D_n) M_n;

$M_1 + M_2 + \ldots + M_n$

Атрибуты: массивы M_1, M_2, . . . M_n, тип элементов массива T, типы размерностей D_1, D_2, . . . D_n.

Правило: объединяемые массивы должны иметь совпадающие типы элементов и непересекающиеся типы индексов, причем типы индексов должны принадлежать некоторому общему типу. Результатом операции является массив. Тип индексов результирующего массива есть объединение типов индексов операндов.

Условие корректности:

$\forall t.\ (t \in D_1) \vee (t \in D_2) \vee \ldots \vee (t \in D_n)$, t принадлежит некоторому общему типу для типов D_1, D_2, . . . D_n

$\forall t.$ **not** $((t \in D_i) \,\&\, (t \in D_j))\ \ \forall i \mathrel{!}= j$

Имя леммы: ArrayUnion*номер*

3.12 Поле объединения

Тип объединения представляется в виде:

union (<конструкторы объединения>)

<конструктор объединения> ::= <имя> [(<описание полей>)]

Пример объединения:

```
type int_tree = union (
  leaf (int val),
  node (int_tree lhs, int v, int_tree rhs)
);
```

Значением типа объединения является значение одного из конструкторов, перечисленных в списке описаний конструкторов.

Конструкция:

```
type S (…) = union (
…
    A (… T: c, …) //A – конструктор, c – поле объединения с типом T.
…
);
S s; //описание переменной типа объединения
... Имеется конструкция s.c ...
```

Атрибуты: тип объединения S, конструктор, поле c, T_c – тип c.
Правило: Надо проверить, что s соответствует конструктору A.
Условие корректности:

Пусть A? – распознаватель, соответствующий конструктору A . Тогда условие корректности: A?(s). Распознаватель является функцией типа **bool** от некоторого выражения. Значение распознавателя истинно, если значением выражения является конструктор с именем A.
Имя леммы: Union*номер*

4. КОНТРОЛЬ ДИНАМИЧЕСКОЙ СЕМАНТИКИ В PVS: ОБЗОР

Способ контроля динамической семантики для языка P во многом аналогичен проводимой в системе PVS, где нетривиальный семантический контроль осуществляется через доказательство формул корректности.

Семантический анализатор системы PVS анализирует теории на семантическую корректность, вычисляя семантические атрибуты во внутреннем представлении, построенном синтаксическим анализатором. Правила семантики определяют условия совместимости типа выражения с типом его позиции внутри объемлющей конструкции. Эти условия называются условиями корректности типов (type-correctness conditions – TCCs) и представляются в виде лемм на языке спецификации PVS. TCC присоединяются к теории и должны быть доказаны.

4.1. Посылки в леммах

Посылки в леммах, генерируемые для TCC, получаются из выражений, включающих IF-THEN-ELSE, AND, OR и IMPLIES. Далее приводится таблица, показывающая, какие посылки генерируются в различных конструкциях.

Expression	Context for E
IF A THEN E ELSE C ENDIF	A
IF A THEN B ELSE E ENDIF	NOT A
A AND E	A
A OR E	NOT A
A IMPLIES E	A

Далее представлены различные виды условий корректности, генерируемых при семантическом анализе.

4.2. TCC существования

PVS допускает пустые типы. Объявленные константы некоторого типа – это элементы этого типа, поэтому тип должен быть непустым, иначе возникает противоречие. Таким образом, когда объявляется константа, система проверяет непустоту типа, и если не может это определить, то генерирует TCC существования.

4.3. Рекурсивные определения

Любая функция в PVS должна быть определена для всех значений аргументов. В описание рекурсивно определяемой функции должна входить дополнительная функция, называемая *мерой*, сигнатура которой совпадает с рекурсивной функцией, но с диапазоном области отношения порядка (по умолчанию < на натуральных числах или ординалах). Отношение порядка должно быть вполне упорядочено, т.е. всякое непустое подмножество должно иметь минимальный элемент. Генерируется TCC вполне упорядоченности.

Пример:

```
factorial(x: nat): RECURSIVE nat =
IF x = 0 THEN 1 ELSE x * factorial(x – 1) ENDIF
MEASURE (LAMBDA (x: nat): x)
```

TCC:
factorial_TCC2: OBLIGATION
FORALL (x: nat): NOT x = 0 IMPLIES x – 1 < x

4.4. TCC подтипа

Имеются операции, определенные на подтипах, которые применяются к элементам родительского типа. Нужно проверить, что данное применение корректно.

Например, деление в PVS – общая функция, чья сигнатура
[real, nonzero real -> real].
Div_form: FORMULA (FORALL (x,y: int): x /= y
IMPLIES (x – y)/(y – x) = -1)

Делитель имеет целый тип, но сигнатура операции / запрашивает вещественный тип без нуля.
TCC:
div_form_TCC1: OBLIGATION
(FORALL (x,y: int): x /= y IMPLIES (y – x) /= 0)

4.5. TCC несогласованности областей определения (domain mismatch)

При утверждении равенства двух функций должно выполняться равенство их типов (т.е. включение в обе стороны). Тип функции $[t_1,\ldots,t_n \to t]$ есть подтип типа $[s_1,\ldots,s_m \to s]$, если t – подтип s, n = m и $s_i = t_i$ для i =1,…,n. Генерируются TCC несоответствия областей.

Пример:
p,q: pred[int]
f: [{x: int | p(x)} -> int]
g: [{x: int | q(x)} -> int]
h: [int -> int]
eq1: FORMULA f = g
eq2: FORMULA f = h

TCC:
eq1_TCC1: OBLIGATION
(FORALL (x1: {x : int | q(x)}, y1 : {x : int | p(x)}) : q(y1) AND p(x1))
eq2_TCC1: OBLIGATION
(FORALL (x1: int, y1 : {x : int | p(x)}) : TRUE AND p(x1))

4.6. ТСС непересечения и покрытия

Конструкция COND – это расширение конструкции IF-THEN-ELSE с набором возможных путей. Имеет вид:

```
COND
be_1 -> e_1,
be_2 -> e_2,
…
be_n -> e_n
ENDCOND
```

Здесь be_i – логический выражения, e_i – выражения общего родительского типа. Должно выполняться условие попарного непересечения условий и того, что их объединение всегда истинно: генерируются ТСС непересечения и покрытия. Если есть альтернатива по умолчанию ELSE, то ТСС покрытия не нужно.

Если be_i – равенства с одинаковой левой частью, а в правой находятся только числа и основные арифметические операции (+, -, *, /), то семантический анализатор непосредственно проверяет непересечение и покрытие, и ТСС не генерируется.

4.7. ТСС оператора CASE

Для оператора CASE в случае отсутствия альтернативы по умолчанию генерируется cases ТСС, которое утверждает, что выражение не может принимать неперечисленных значений.

5. РЕАЛИЗАЦИЯ В СИСТЕМЕ ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

Описывается структура теорий, генерируемых в процессе семантического анализа программы, и алгоритм генерации условий семантической корректности.

5.1. Теории модуля

Предикатная программа составлена из модулей. Модуль является единицей компиляции и содержит набор глобальных описаний и определения предикатов. Глобальные объекты модуля (типы, константы, формулы, константы), используемые в генерируемых теориях, формируют главную теорию во внутреннем представлении.

Для каждого определения предиката подсистема контроля динамической семантики строит теорию, имя которой совпадает с именем предиката. Набор теорий, создаваемый генератором формул корректности, состоит из главной теории и теорий, сгенерированных для определений предикатов. Набор теорий должен быть замкнутым: для всякого вхождения имени (формулы, структурного типа, переменной, …) имеется соответствующее описание за исключением имен примитивных типов.

Теория состоит из формул и утверждений. Локальные объекты (типы, константы, переменные), используемые в формулах и утверждениях, должны быть представлены как объекты генерируемого модуля. Глобальные объекты должны быть определены в других теориях, и эти теории должны присутствовать в списке импорта модуля. Если локальный объект строится на базе другого локального объекта, то последний должен быть определен ранее. Во внутреннем представлении теория кодируется конструкцией МОДУЛЬ

5.2. Теория для определения предиката

Для определения предиката

A(x**:** y) **pre** P(x) { S(x**:** y) } **post** Q(x, y)

генерируется следующая теория, представленная модулем:

module A;

<описания локальных объектов>

formula P_A(x) = P(x);

<леммы с формулами контроля динамической семантики оператора S>

Здесь P_A – новое имя, образованное из имени A. Это имя должно отличаться от других имен, используемых в программе.

Для определения предиката-гиперфункции

A(x**:** y #1**:** z #2) **pre** P(x); **pre** 1: C(x) { W } **post** 1: S(x, y); **post** 2: R(x, z)

генерируется следующая теория:

module A;

<описания локальных объектов>

<список локальных типов>

formula P_A(x) = P(x);

formula P1_A(x) = C(x);

< леммы с формулами контроля динамической семантики оператора W>

5.3. Общее описание алгоритма генерации условий семантической корректности

По дереву предикатной программы делается обход. Для каждого предиката создается теория в виде модуля с именем этого предиката. Его предусловия вставляются в модуль в виде формул. Для конструкций, встречающихся в теле предиката, генерируются условия семантической корректности на языке внутреннего представления и записываются в виде лемм в модуль.

При встрече конструкции, для которой необходимо сгенерировать лемму, применяется метод, собирающий конъюнкциями предусловие предиката и условия в условных операторах. Так составляется посылка для леммы. Утверждение леммы генерируется по определенным выше алгоритмам.

Например, при встрече вхождения элемента массива:

```
type Ar(int n) = array(int, 1..n);
P(int k, Ar M(k): int m)
{     …
      m = M[i];
      …
}
```

В конструкции M[i] (являющейся деревом), есть, кроме прочего, индекс i и диапазон массива M. Из них генерируется выражение на языке внутреннего представления (1 <= i & i <= k). Далее, используя полученное выражение D из взятия условий из условных операторов и предусловия, генерируется лемма с утверждением D => (1 <= i & i <= k).

В большинстве конструкций леммы генерируются именно таким образом: взятием из конструкции необходимых элементов и составлением их них формулы.

Сложной задачей является генерация условия совместимости двух типов. Семантический анализатор уже проверил структурную совместимость этих типов, и нужно проверить совместимость этих типов, если у них есть параметры (в данный момент это подтипы, массивы, диапазоны и предикаты). Для этого делаются обходы этих типов (представляемых деревьями), и при встрече параметризованных конструкций внутри этого типа запоминаются пути (последовательность отцов) до них. Такие конструкции являются листьями деревьев, и если пути до двух таких конструкций из соответствующих сравниваемых деревьев совпадают, то это означает, что они находятся на соответствующих местах в типах, и их необходимо проверить на совместимость (алгоритм описан в разд. 3).

6. ТРАНСЛЯЦИЯ ТЕОРИЙ НА ЯЗЫК СИСТЕМЫ PVS

Сгенерированные теории необходимо транслировать в систему автоматического доказательства PVS.

Во внутреннем представлении теория кодируется конструкцией МОДУЛЬ. Теория во внутреннем представлении состоит из объявлений типов, формул и утверждений.

module M;
type A = …
type B = …
formula a(…) = …
formula b(…) = …
lemma …
lemma …

Обозначим tr – отображение конструкций из языка P в язык спецификаций системы PVS. Описание образов конструкций реализуется в следующей форме:

tr(<языковая конструкция языка P>) = <образ конструкции в PVS>

Образом модуля в PVS является следующая конструкция:

tr (module A; …) =
M : THEORY
BEGIN
<объявление переменных>
tr(<описание формул>)
tr(<описание лемм>)
END M

6.1. Объявление переменных

tr(<имя типа> <идентификатор>)
= tr(<идентификатор>): VAR tr(<имя типа>)

Описание параметра предиката или формулы выносится выше. В списке параметров имена параметров используются без описания. Такой способ допускается в языке спецификаций PVS.

a: VAR nat
P (a) : …

Ввиду этого задача трансляции идентификаторов усложняется. Рассмотрим пример:
formula f1(**nat** a) = …
formula f2(**bool** a) = …

В случае попытки вынести объявление переменных за пределы формальных параметров формулы возникнет конфликт: две переменные с одним именем, но разных типов. Для разрешения подобных конфликтов был реализован т.н. mangling – искажение имен идентификаторов, с целью придания им уникальности.
tr(<идентификатор>) = <идентификатор>_<mangling>

Так будет выглядеть блок-объявление переменных для данного примера:
a_n: VAR nat
a_b: VAR bool

6.2. Трансляция констант и переменных

Образом констант являются сами константы.
tr (<константа>) = <константа>

Если в качестве переменной выступает только её идентификатор, образом будет этот же идентификатор, но с учетом искажения.
tr(<идентификатор>) = <идентификатор>_<mangling>

Массив в системе PVS представляет собой функцию от индексов, поэтому, если в качестве переменной выступает элемент массива, то его образ:
tr(<выражение>[<список индексов>]) =
tr(<выражение>)(tr(<список индексов>))

Если в качестве переменной выступает элемент поля, то его образ будет следующим:
tr(<выражение>.<имя поля>) = tr(<выражение>) ’ <имя поля>

Мультипеременная и мультивыражение отображаются следующим образом:
tr(|<список выражений>|) = (tr(<список выражений>))

6.3. Трансляция унарных и бинарных выражений

Для унарной op a и бинарной a op b операции, где op – операция, a и b – выражения, реализуется отображения:

tr(op a) = tr(op) (tr(a))
tr(a op b) = (tr(a)) tr(op) (tr(b))

Операнды a и/или b обрамляются круглыми скобками после их перевода на язык PVS.

Отображения унарных операций следующие:

tr(^) = ^
tr(+) – унарный плюс отсутствует
tr(–) = -
tr(!) = NOT

Отображения бинарных операций следующие:

tr(*) = *
tr(/) = /
tr(a % b) = rem(b)(a)
tr(+) = +
tr(x **in** a) = member(x, a)
tr(<) = <
tr(>) = >
tr(<=) = <=
tr(>=) = >=
tr(=) = =
tr(!=) = /=
tr(&) = & для логических операндов
tr(a & b) = intersection(a, b) для множеств
tr(a **xor** b) = a XOR b для логических
tr(a **xor** b) = symmetric_difference(a, b) для множеств
tr(a **or** b) = a OR b для логических
tr(a **or** b) = union(a, b) для множеств
tr(=>) = =>
tr(<=>) = <=>

6.4. Трансляция кванторных выражений

Образ кванторного выражения определяется следующим образом:

tr(<квантор> <список переменных>. <выражение>)
= tr(<квантор>) tr(<список переменных>) : tr(<выражение>)

Образы кванторов следующие:

tr(forall) = FORALL
tr (exists) = EXISTS

6.5. Трансляция типов

Отображение примитивных базисных типов тождественно, поскольку в PVS используются те же имена:

tr(**bool**) = bool
tr(**nat**) = nat
tr(**int**) = int
tr(**real**) = real
tr(**char**) = char

Для имени типа из другого модуля предикатной программы:

tr(<имя модуля>.<имя типа>) = tr(<имя модуля>) @ tr(<имя типа>)

Для имени типа с параметрами:

tr(<имя типа>(<аргументы>)) = tr(<имя типа>) (tr(<аргументы>))

Для определения имени типа как параметра предиката или описания другого типа:

tr(**type** <имя типа>) = tr(<имя типа>) : TYPE

Отображение конструкции подмножества типа реализуется следующим образом:

tr((<изображение типа> <имя переменной>) : <выражение>)) =
{ tr(<имя переменной>) : tr(<изображение типа>) | tr(<выражение>) }

Для изображения диапазона:

tr(<выражение1> .. <выражение2>) =
subrange(tr(<выражение1>), tr(<выражение2>))

Для описания типа перечисления:

tr(**type** <имя типа> = **enum** (<список имен>)) = enum: TYPE = {
tr(<список имен>) }

Для изображения типа структуры:

tr(**structure** (<описание полей>)) = [# tr(<описание полей>) #]

Образ изображения списка описания полей имеет следующую структуру:

tr(<изображение типа> <список имен полей>) =
tr(<список имен полей>) : tr(<изображение типа>)

Строковый тип представляется иначе, чем тип string в PVS:

tr(**string**) = list[char]

Для изображения типа множества:

tr(**set** (<изображение базового типа>)) =
setof[tr(<изображение типа>)]

Для различных операций с множествами:

tr(~ <выражение>) = complement(tr(<выражение>))

tr(<выражение1> + <выражение2>) = union(tr(<выражение1>),
tr(<выражение2>))
tr(<выражение1> **or** <выражение2>) = union(tr(<выражение1>),
tr(<выражение2>))
tr(<выражение1> – <выражение2>) =
difference(tr(<выражение1>), tr(<выражение2>))
tr(<выражение1> & <выражение2>) =
intersection(tr(<выражение1>), tr(<выражение2>))
tr(<выражение1> **in** <выражение2>) =
member(tr(<выражение1>), tr(<выражение2>))

Для операций **len**, **bits**, **mask** нет прямых аналогов в библиотеках PVS.

6.7. Трансляция массивов

Для изображения типа массива:
Пример на языке P:
array (**real**, 1..k, 1..k+1);
tr(**array** (<изображение типа элемента>, <типы измерений массива>) =
[tr(<типы измерений массива>) ->tr(<изображение типа элемента>)].

Типы измерений массива – перечисление типов индексов по каждому измерению.

Для многомерного массива тип индексов (исходного массива и суженного) отображается как tuple.

Для определения массива:
пример на языке P:
for (**var** i) { **case** 1..10 : i+1 **case** 11..19 : 2*i **case** 20 : -1 **default** : 0 }
tr(**for** (<задание индексов>)<тело определения массива>) =
(LAMBDA tr(<задание индексов>) : tr(<тело определения массива>))
tr(<задание индексов>) =
(tr(<определение индекса1>)), ..., (tr(<определение индексаN>))
tr(<определение индекса>) =
tr(<индекс>) : tr(<изображение типа индекса>))
tr(<тело определения массива>) =
tr(<выражение для определения массива>) |
COND
tr(<индексы части1>) ->
tr(<выражение для определения массива>) ,
...
tr(<индексы частиN>) ->

tr(<выражение для определения массива>)
[ELSE -> tr(<выражение для определения массива>)]
ENDCOND

Конструкция “модификация массива” реализуется как определение массива, в котором добавляется ELSE-часть вида A[i], где A – модифицируемый массив, а i – набор индексов в **for**–части.

tr(<индексы части>) =
(tr(<набор индексов1>) AND (tr(<набор индексов2>) AND …
AND (tr(<набор индексовM>))

tr(<набор индексов>) =
tr(<значение или диапазон1>, j) OR … OR tr(<значение или диапазонK>, j)

Здесь j — индекс по соответствующему измерению в **for**–части.

6.8. Трансляция формул и лемм

tr(**formula** <имя формулы>(<описание формальных параметров>:
<имя типа результата>) = <выражение>) =
<имя формулы> (tr(<описание формальных параметров>)):
tr(<имя типа результата>) = tr(<выражение>)

tr(<тип параметра 1> <идентификатор 1>,
<тип параметра 2> <идентификатор 2>, …) =
tr(<идентификатор 1>), tr(<идентификатор 2>), …

tr(**lemma** <выражение>) = L<номер леммы>: LEMMA tr(<выражение>)

7. ДОКАЗАТЕЛЬСТВО УТВЕРЖДЕНИЙ В СИСТЕМЕ PVS

PVS – это система верификации с языком спецификаций высокого уровня и мощным блоком доказательства. Схема работы: проводится синтаксический, затем семантический анализ, и доказываются утверждения.

Возможна интеграция PVS с SMT-решателем Yices, который может доказывать формулы определенных видов, что увеличивает возможности доказательства.

SMT (Satisfiability Modulo Theories) – это система автоматической разрешимости ограниченного класса формул на базе теории, описывающей класс формул. Примерами таких теорий для SMT формул являются теории целых и вещественных чисел, теории списков, массивов, битовых векторов и т.д.

Yices – эффективный SMT решатель (автоматически строит доказательство), определяющий выполнимость произвольной формулы, содержащей неинтерпретируемые функциональные символы равенства, линейные целые и действительные числа арифметики, скалярные типы, рекурсивные типы данных, тупли, записи, массивы, фиксированного размера битвекторы, кванторы и лямбда выражения.

Пример 1

Программа на языке предикатного программирования:

```
Умн(nat a, b:  nat c)
pre a ≥ 0 & b ≥ 0
{
      if (a = 0) c = 0
      else c = b + Умн(a – 1, b)
}
post c = a ∗ b measure a;
```

Теория после семантического анализа:

```
module
formula P(nat a, b) = a >= 0 & b >= 0;
lemma P(a, b) & not a = 0 => a – 1 >= 0;
end
```

После трансляции в PVS:

```
% Declarations.
a_n, b_n: VAR nat
% Formulas.
P (a_n, b_n) : bool = (a_n >= 0) AND (b_n >= 0)
% Lemmas.
L1: LEMMA (P(a_n, b_n) AND (NOT a_n = 0)) IMPLIES (a_n – 1 >= 0)
```

Пример 2

Программа на языке предикатного программирования:

```
type Vec1(nat n) = array (real, 1..n);
type Vec2(nat n) = array (real, n+1..2*n);

sum( nat n : )
{
      Vec1(n) v1 = for (var i) { case 1..n : 0};
```

```
    Vec2(n) v2 = for (var i) { case n+1..2*n: 1};
    Vec1(2*n) v;
    v = v1 + v2;
}
```

Теория после семантического анализа для объединения массивов:

module
lemma !(((1<=i) & (i<=n)) & ((n+1<=i) & (i<=2*n)));
lemma ((1<=i) & (i<=2*n)) ⇔ (((1<=i) & (i<=n)) **or**
((n+1<=i) & (i<=2*n)));
end

После трансляции в PVS:

% Declarations.
i_n, n_n: **VAR** nat
% Lemmas.
L1: **LEMMA NOT** (((1<= i_n) **AND** (i_n <= n_n)) **AND**
((n_n +1<= i_n) **AND** (i_n <=2* n_n)))
L2: **LEMMA** ((1<= i_n) **AND** (i_n <=2* n_n)) **IFF** (((1<= i_n) **AND** (i_n <= n_n)) **OR** ((n_n +1<= i_n) **AND** (i_n <=2* n_n)));

Доказательство утверждений на PVS

Имеется набор команд для интерактивного доказательства, а также общие стратегии доказательства. Леммы, сгенерированные по условиям семантической корректности, строго заданного вида, причём для каждого вида можно подобрать свою стратегию доказательства.

Доказательство на PVS представляется деревом. Утверждение можно разбить на части и доказывать части отдельно (таким образом, дерево доказательства ветвится). Вершины дерева – выполненные команды доказательства.

Стратегия grind, как правило, хороший способ доказательства, если не требуется индукции, и требуется только раскрытие определений, арифметических приведений, равенств и кванторов.

8. ЗАКЛЮЧЕНИЕ

В настоящей работе описывается контроль динамической семантики предикатной программы с помощью системы автоматического доказательства PVS.

Разработаны формальные условия корректности динамической семантики и алгоритмы их построения для всех конструкций языка P. Реализован генератор условий корректности динамической семантики для большинства конструкций языка P. Генератор является back-end'ом системы предикатного программирования.

Реализован конвертор, позволяющий автоматически переносить полученные условия корректности на язык спецификаций системы PVS. Конвертор является частью системы предикатного программирования и может быть вызван автоматически.

Программа генератора тестировалась на 15 тестовых предикатных программах.

Планы дальнейших работ следующие: сгенерировать условия корректности для задачи совместимости типов, для оператора switch для случая отсутствия альтернативы по умолчанию, для полей объединения и доказать их на PVS; конвертор с P на PVS переводит пока только простые типы. Планируется также построить стратегии, вызывающие Yices при доказательстве формул на PVS.

СПИСОК ЛИТЕРАТУРЫ

1. Шелехов В.И. Введение в предикатное программирование. — Новосибирск, 2002. – 82с. – (Препр. / ИСИ СО РАН; № 100).
2. Шелехов В.И. Язык предикатного программирования P. – Новосибирск, 2002. – 40с. – (Препр. / ИСИ СО РАН; № 101).
3. PVS Specification and Verification System. – SRI International. http://pvs.csl.sri.com/documentation.shtml
4. S. Owre, N. Shankar, J. M. Rushby, D. W. J. Stringer-Calvert. PVS Language Reference http://pvs.csl.sri.com/documentation.shtml
5. S. Owre, N. Shankar, J. M. Rushby, D. W. J. Stringer-Calvert. PVS System Guide http://pvs.csl.sri.com/documentation.shtml
6. Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, William Pugh, "Using Static Analysis to Find Bugs," IEEE Software, vol. 25, no. 5, pp. 22-29, Sep./Oct. 2008.
7. P. Cousot, MITcourse "Abstract interpretation", http://web.mit.edu/16.399/www/, Feb.-May, 2005.
8. Dutertre B., Moura L. The YICES SMT solver. —2006. http://yices.csl.sri.com/tool-paper.pdf