АУДК 004.415.52

Синтез операторов предикатной программы

В.И. Шелехов

Институт систем информатики им. А.П. Ершова, г. Новосибирск,
e-mail: vshel@iis.nsk.su

Рассматривается программный синтез отдельных операторов предикатной программы в интеграции с дедуктивной верификацией и обычным стилем конструирования программы в редакторе Eclipse. Анализируются методы синтеза операторов на примере эффективной программы сортировки методом простых вставок. Предлагается архитектура интегрированной системы дедуктивной верификации и программного синтеза.

Ключевые слова: формальная операционная семантика, программный синтез, дедуктивная верификация, SMT-решатель, система автоматического доказательства PVS, сортировка.

Введение

Язык предикатного программирования P [1] является языком доказательного программирования. Предикатная программа H(x**:** y) с аргументами x и результатами y задается вместе с ее спецификацией: предусловием P(x) и постусловием Q(x, y). Программный синтез есть автоматический процесс построения программы H(x**:** y) по ее спецификации.

Исследования в области синтеза программ имеют длительную историю. Используются разные методы. *Конструктивный* программный синтез [2] предполагает извлечение программы посредством унификации из конструктивного доказательства теоремы существования: $\forall x.\ P(x) \Rightarrow \exists y.\ Q(x, y)$. В *дедуктивном* синтезе программа, представляемая в вычислимой логике, выводится из ее спецификации. *Схемный* синтез реализуется на базе набора образцов (схем) программ. *Синтаксически-управляемый* синтез реализуется не для полной программы, а для некоторых позиций в синтаксических конструкциях программы [3]. Используются также методы *обучения* [4].

Система синтеза и верификации Leon [5, 6] позволяет по спецификации автоматически за несколько секунд синтезировать рекурсивную функциональную программу до 7 строк. Программа строится перебором множества заготовок программ на базе эффективного механизма нумерации термов. Отметим, что перебор

относительно небольшой, поскольку разрешается использовать только алгебраические типы данных. Пустые позиции в заготовках сначала определяются из контрпримеров, получаемых от SMT-решателя Z3, а затем инкрементально уточняются применением логической абдукции и ряда других методов. Эффективная отбраковка вариантов программ реализуется автоматической генерацией тестов и их прогоном, используя быструю упрощенную версию транслятора с языка Scala. Успех данного подхода был бы невозможен без кардинального улучшения возможностей SMT-решателей, в частности, реализации в них методов автоматического доказательства по индукции.

В настоящее время, автоматический синтез программ по формальным спецификациям, несмотря на значительные достижения в этой области, возможен лишь для простых коротких программ.

В нашей работе [7] рассматривается задача синтеза небольших фрагментов предикатной программы. Применяются правила синтеза, являющиеся частью ранее разработанной системы правил доказательства корректности для дедуктивной верификации предикатных программ [8]. Синтез фрагментов реализуется в интеграции с дедуктивной верификацией [8, 9, 14] в среде редактора Eclipse. Фрагменты программы, которые программист написал или модифицировал в редакторе Eclipse, автоматически проверяются на корректность подсистемой дедуктивной верификации. Фрагменты, отмечаемые в тексте комбинацией «[*]», программист поручает сгенерировать подсистеме синтеза, запускаемой в параллельном режиме. Однако синтез фрагментов даже простых программ часто блокируется ввиду невозможности используемых SMT-решателей CVC3 и Z3 разрешить формулы корректности, поставляемые от правил синтеза.

Настоящая работа продолжает исследование методов синтеза предикатных программ, причем синтезируемый фрагмент ограничен одним простым оператором. В разделе 1 дается краткое описание языка предикатного программирования P [1].

Используемые методы дедуктивной верификации и программного синтеза изложены в разделе 2. Задача синтеза исследуется в разделе 3 на примере вполне реалистичной эффективной программы сортировки массивов методом простых вставок. Отметим наличие значительного числа работ по синтезу программ сортировки списков, например, [10]. Однако с помощью списков проблематично описать эффективные алгоритмы сортировки, в частности, самый быстрый алгоритм сортировки [9]. В заключении формулируется утверждение о принципиальной невозможности автоматического синтеза реальных программ и предлагается архитектура интерактивной системы дедуктивной верификации и программного синтеза.

1. Предикатное программирование

Предикатная программа относится к классу программ-функций [11] и является предикатом в форме вычислимого оператора H(x**:** y) с аргументами x и результатами y. Минимальный полный *базис предикатных программ* определен в виде языка $\mathbf{P_0}$. Предикатная программа определяется следующей конструкцией:

<имя предиката>(<аргументы>**:** <результаты>) { <оператор> }

Пусть x, y и z обозначают разные непересекающиеся наборы переменных. Набор x может быть пустым, наборы y и z не пусты. В составе набора переменных x может использоваться логическая переменная e со значениями **true** и **false**. Пусть B и C – имена предикатов, A и D – имена переменных предикатного типа. Операторами являются: *оператор суперпозиции* B(x**:** z); C(z**:** y), *параллельный оператор* B(x**:** y) || C(x**:** z), *условный оператор* **if (**e) B(x**:** y) **else** C(x**:** y), *вызов предиката* B(x**:** y) и *оператор каррирования*. В таблице 1 представлен полный базис вычислимых предикатов и соответствующих им операторов.

H(x: y) ≅ ∃z. B(x: z) & C(z: y)	H(x: y) { B(x: z); C(z: y) }
H(x: y, z) ≅ B(x: y) & C(x: z)	H(x: y, z) { B(x: y) || C(x: z) }
H(x: y) ≅ (e ⇒ B(x: y)) & (¬e ⇒ C(x: y))	H(x: y) { **if** (e) B(x: y) **else** C(x: y) }
H(x: y) ≅ B(x˜ : y)	H(x: y) { B(x˜: y) }
H(A, x: y) ≅ A(x: y)	H(A, x: y) { A(x: y) }
H(x: D) ≅ ∀y,z. D(y: z) ≡ B(x, y: z)	H(x: D) { D(y: z) {B(x, y: z)} }
H(A, x: D) ≅ ∀y,z. D(y: z) ≡ A(x, y: z)	H(A, x: D) { D(y: z) {A(x, y: z)} }

Таблица 1. Вычислимые предикаты и их программная форма

Набор x˜ составлен из набора переменных x с возможным добавлением имен предикатных программ.

Имеется два вида оператора каррирования: D(y: z) {B(x, y: z)} и D(y: z) {A(x, y: z)}. Результатом исполнения оператора каррирования является новый предикат D, получаемый фиксацией значения набора x.

На базе языка **P₀** последовательным расширением [15] построен язык предикатного программирования **P** [1]. В языке **P** нет циклов и указателей; вместо них используются рекурсивные программы и алгебраические типы данных.

Определим программу with модификации результата предиката A для значения аргумента x = i через оператор каррирования:

with(A, i, z: D) { D(x: y) { **if** (x = i) y = z **else** A(x: y)} } .

Вызов программы with(B, i, z: C) будем записывать в виде C = B **with** (i: z).

Эффективность предикатных программ достигается применением следующих оптимизирующих трансформаций [15], переводящих программу на императивное расширение языка **P**:

- замена хвостовой рекурсии циклом;
- подстановка тела программы на место ее вызова;
- склеивание переменных: замена всех вхождений одной переменной на другую переменную;
- кодирование алгебраических типов (списков и деревьев) с помощью массивов и указателей.

2. Дедуктивная верификация и программный синтез

Операционную семантику программы H(x: y) определим в виде предиката:

$\mathcal{R}(H)(x, y) \cong$ для значения набора x исполнение программы H всегда завершается и существует исполнение программы, при котором результатом вычисления является значение набора y.

Для языка $\mathbf{P_0}$ построена формальная операционная семантика и доказано тождество $\mathcal{R}(H) = H$ [12]. На базе языка $\mathbf{P_0}$ последовательным расширением и сохранением тождества $\mathcal{R}(H) = H$ построен язык предикатного программирования **P** [1].

Спецификацией программы $H(x\mathbf{:}\ y)$ являются два предиката: *предусловие* $P(x)$ и *постусловие* $Q(x, y)$. Спецификация записывается в виде: $[P(x), Q(x, y)]$. *Тотальная корректность* программы относительно спецификации определяется формулой:

$$H(x\mathbf{:}\ y)\ \mathbf{corr}\ [P(x), Q(x, y)] \cong \forall x.\ P(x) \Rightarrow [\forall y.\ \mathcal{R}(H)(x, y) \Rightarrow Q(x, y)\ ]\ \&\ \exists y.\ \mathcal{R}(H)(x, y)$$

Ввиду тождества $\mathcal{R}(H) = H$ формула тотальной корректности принимает вид:

$$P(x)\ \&\ H(x\mathbf{:}\ y) \Rightarrow Q(x, y) \qquad (1)$$

для предикатов H, удовлетворяющих условию тотальности: $P(x) \Rightarrow \exists y.\ H(x\mathbf{:}\ y)$.

Для основных операторов (параллельного, условного и суперпозиции) разработана система правил доказательства их корректности [8], в том числе и при наличии рекурсивных вызовов, существенно упрощающая процесс доказательства по сравнению с исходной формулой тотальной корректности (1). Корректность правил доказана [16] в системе PVS. В системе предикатного программирования реализован генератор формул корректности программы. Значительная часть формул доказывается автоматически SMT-решателем CVC3. Оставшаяся часть формул генерируется для системы интерактивного доказательства PVS [13].

Предположим, что наборы переменных x, y и z не пересекаются, а x может быть пустым. Ниже приведены некоторые правила доказательства корректности операторов.

$$\mathbf{QP:}\ \frac{B(x\mathbf{:}\ y)\ \mathbf{corr}\ [P(x), Q(x, y)];\ C(x\mathbf{:}\ z)\ \mathbf{corr}\ [P(x), R(x, z)];}{\{B(x\mathbf{:}\ y)\ ||\ C(x\mathbf{:}\ z)\}\ \mathbf{corr}\ \ [P(x), Q(x, y)\ \&\ R(x, z)]}$$

$$\textbf{QC:}\ \frac{\begin{array}{l}B(x\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ E(x),\ Q(x,\ y)];\\ C(x\textbf{:}\ z)\ \textbf{corr}\ [P(x)\ \&\ \neg E(x),\ Q(x,\ y)]\end{array}}{\{\textbf{if (}E(x))\ B(x\textbf{:}\ y)\ \textbf{else}\ C(x\textbf{:}\ y)\}\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Далее следует правило для частного случая оператора суперпозиции, соответствующего *сведению к более общей задаче* C(x, z**:** y).

$$\textbf{RBE:}\ \frac{\begin{array}{l}\forall z\ C(x,\ z\textbf{:}\ y)\ \textbf{corr*}\ [P_C(x,\ z),\ Q(x,\ y)];\\ P(x) \rightarrow\ \exists z.\ B(x\textbf{:}\ z)\ \&\ P_C{}^*(x,\ B(x));\end{array}}{C(x,\ B(x)\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Запись вида z = B(x) является эквивалентом B(x**:** z). Истинность двух посылок правила **RBE** гарантирует корректность следующей программы:

H(x**:** y) **pre** P(x) **post** Q(x, y) { C(x, B(x)**:** y) }

В случае рекурсивного вызова C(x, B(x)**:** y) обозначение **corr*** означает, что первая посылка опускается, а $P^*_C(x)$ заменяется на $P_C(x,\ B(x))\ \&\ m(x) < m(v)$. Здесь m – натуральная функция *меры*, строго убывающая на аргументах рекурсивных вызовов, а v обозначает аргументы рекурсивной программы C.

Задача *программного синтеза* заключается в нахождении тотального предиката H на языке **P**, обеспечивающего истинность формулы $P(x)\ \&\ H(x\textbf{:}\ y) \Rightarrow Q(x,\ y)$. Вместо данной формулы можно использовать любое из правил корректности и пытаться искать такие подоператоры B или C, чтобы все посылки правила стали истинными. Это гарантирует корректность оператора, указанного в заключении правила.

3. Синтез программы сортировки простыми вставками

Разработка программы на языке **P** в стиле синтеза начинается с построения базисных теорий для типов данных исходной задачи. Введем теорию SortDecl для определения типа Arn массива элементов произвольного типа T. Для типа T должно быть определено отношение "≤" линейного (или тотального) порядка. Индексы массива определены типом natn, являющимся диапазоном от 0 до некоторого натурального n.

```
theory SortDecl{
   type T("≤", "<", "≥", ">");
   import Total_order(T);
   nat n;
   type natn = 0..n;
   type Arn = array (T, natn);
};
```

Спецификация программы сортировки: итоговый массив должен быть отсортирован и получен из исходного массива перестановкой элементов. Свойство перестановочности определяется следующей теорией.

```
theory Perm {
   import SortDecl;
   type F = subtype (predicate (natn: natn) f:  bijective(f));
   formula perm(Arn a, b) = ∃ F f. ∀ natn j. a[j] = b[f(j)];
   Arn a, b, c;
   pe1: lemma perm(a, a);
   pe2: lemma perm(a, b) & perm(b, c) ⇒ perm(a, c);
};
```

Предикат perm определяет перестановочность массивов a и b в случае существования биективной (взаимно-однозначной) функции, отображающей элементы массива b в элементы массива a. Леммы pe1 и pe2 определяют свойства рефлексивности и транзитивности, которые наверняка потребуются в доказательствах.

Свойство сортированности определено в теории Sort предикатом sorted: следующие элементы не могут быть меньше предыдущих.

```
theory Sort {
   import SortDecl;
   formula sorted(Arn a) =  ∀ natn i, j. i < j ⇒  a[i] ≤ a[j];
};
```

Формальная спецификация программы сортировки представлена в модуле SORT.

```
module SORT {
   import Sort, Perm;
   sort(Arn a: a') post perm(a, a') & sorted(a')
end SORT;
```

Имя a′ подразумевает, что в реализации переменная a′ должна быть склеена с a. Иначе говоря, отсортированный массив должен быть получен в том же массиве a.

Алгоритм сортировки простыми вставками реализует вставку очередного элемента внутрь уже отсортированной части массива. Предположим, что начальная часть массива a до элемента m отсортирована. Далее требуется вставить элемент a[m+1] внутрь начальной части в диапазоне 0..m таким образом, чтобы итоговый массив стал отсортированным в диапазоне 0..m+1. Свойство сортированности начальной части массива необходимо добавить в теорию Sort.

```
theory Sort { ……..
   formula sorted(Arn a, natn m) =  ∀ i, j = 0..m. i < j ⇒  a[i] ≤ a[j];
   Arn a;
   so1: lemma sorted(a, n) ⇒ sorted(a);
};
```

Модуль SORT1 определяет спецификацию более общей программы сортировки при условии, что часть массива в диапазоне от 0 до m уже отсортирована.

```
module SORT1 {
   import Sort, Perm;
   sort1(Arn a, natn m: Arn a′)
   pre sorted(a, m)
   post perm(a, a′) & sorted(a′);
end SORT1;
```

Синтезируем программу sort сведением к более общей программе sort1, используя правило **RBE**:

$$\textbf{RBE}:\ \frac{\begin{array}{l}\forall\ \text{natn m. sort1(a, m: a′) } \textbf{corr}\ [\text{sorted(a, m)}, \text{Q(a, a′)}];\\ \exists \text{natn m. B(a: m) \& sorted(a, B(a));}\end{array}}{\text{sort1(a, B(a): y) } \textbf{corr}\ [\textbf{true}, \text{Q(a, a′)}]}$$

Здесь Q(a, a′) = perm(a, a′) & sorted(a′).

Задача синтеза – найти предикат B, для которого вторая посылка правила будет истинной, т.е. найти тотальный предикат B, обеспечивающий истинность формулы sorted(a, B(a)). Нахождение требуемого предиката B реализуется перебором термов, зависящих от a. Такими термами могут быть: 0, 1, n, n-1, len(a). При B(a)=0 в формуле sorted(a, 0) кванторная часть вырождается, и она становится тривиально истинной. Истинность sorted(a, 0) может быть установлена обращением к SMT-решателю.

В соответствии с правилом **RBE** результатом синтеза должна быть программа:

```
sort(Arn a: Arn a') post perm(a, a') & sorted(a') { sort1(a, 0: a') };
```

Модуль POP_INTO определяет спецификацию программы pop_into для вставки элемента a[m+1] внутрь отсортированной части массива a в диапазоне индексов от 0 до m. Результат – массив a' с отсортированной частью в диапазоне 0..m+1.

```
module POP_INTO {
   import Sort, Perm;
   pop_into(Arn a, natn m, T e: Arn a')
   pre m < n & sorted(a, m) & e = a[m+1]
   post perm(a, a') & sorted(a', m+1);
end POP_INTO;
```

Далее представим программу sort1 с двумя позициями «[*]», которые должны быть автоматически синтезированы.

```
sort1(Arn a, natn m: Arn a')
pre sorted(a, m) post perm(a, a') & sorted(a')
{  if (m = n) [*]
   else   { pop_into(a, m, a[m+1]: Arn c); sort1([*]) }
};
```

Применим правило **QC** для условного оператора. Посылки правила определяют следующие две цели:

```
[*] corr [sorted(a, m) & m = n, Q(a, a') ]
pop_into(a, m, a[m+1]: Arn c); sort1([*]) corr [sorted(a, m) & m < n, Q(a, a') ]
```

где Q(a, a') = perm(a, a') & sorted(a')

Для первой цели формула корректности (1) переписывается следующим образом:

sorted(a, m) & m = n & X(a, m: a') ⇒ perm(a, a') & sorted(a')

Здесь X(a, m: a') – неизвестный предикат, который следует выбрать так, чтобы эта формула стала истинной. Необходимо независимо обеспечить истинность двух конъюнктов perm(a, a') и sorted(a'). В соответствии с леммой pe1 истинность perm(a, a') обеспечивается использованием оператора присваивания a' = a. Подсистема синтеза должна уметь найти такое решение. Истинность второго конъюнкта следует из леммы so1.

Для второй цели используем следующую модификацию правила **RS**:

RS1:
$$\frac{\begin{array}{l} B(x: z)\ \mathbf{corr}^*\ [P_B(x), Q_B(x, z)];\ \forall z\ C(x, z\mathbf{:}\ y)\ \mathbf{corr}^*\ [P_C(x, z), Q_C(x, z, y)]; \\ \forall z, y\ (\ P(x)\ \&\ Q_B(x, z)\ \&\ Q_C(x, z, y) \rightarrow\ Q(x, y)\ ) \\ \forall z\ (\ P(x)\ \&\ Q_B(x, z) \rightarrow\ P^*_C(x, z)\ ); \\ P(x) \rightarrow\ P^*_B(x); \end{array}}{B(x\mathbf{:}\ z);\ C(x, z\mathbf{:}\ y)\ \mathbf{corr}\ [P(x), Q(x, y)]}$$

Поскольку sort1 рекурсивна, вторая посылка опускается. Правило конкретизируется следующим образом:

RS':
$$\frac{\begin{array}{l} pop\_into(a, m, e\mathbf{:}\ c)\ \mathbf{corr}\ [Ppop(a,m, e), Qpop(a, m, c)]; \\ \forall\ a',c.\ (P(a, m)\ \&\ Qpop(a, m, c)\ \&\ Q(U, Z) \rightarrow\ Q(a, a')\ ) \\ \forall\ c\ .\ (P(a, m)\ \&\ Qpop(a, m, c) \rightarrow sorted(U, V)\ \&\ h(U, V)<h(a, m)); \\ P(a, m)\ \rightarrow\ Ppop(a, m, a[m+1]); \end{array}}{pop\_into(a, m, a[m+1]\mathbf{:}\ c);\ sort1\ (U, V\mathbf{:}\ Z)\ \mathbf{corr}\ [P(a, m), Q(a, a')]}$$

Здесь U, V, Z – переменные, обозначающие неизвестные термы, обеспечивающие истинность посылок правила; Ppop и Qpop – предусловие и постусловие программы pop_into; P(a, m) = sorted(a, m) & m<n; Q(a, a') = perm(a, a') & sorted(a'), h – функция меры на аргументах a и m.

Структура оператора суперпозиции допускает единственные значения для переменных U и Z: U = c, Z = a'. Терм, обозначенный переменной V, зависит от переменных a, m, c.

Посылки 2, 3 и 4, истинность которых надо обеспечить, преобразуются в следующие формулы:

$sorted(a, m)\ \&\ m<n\ \&\ perm(a, c)\ \&\ sorted(c, m+1)\ \ \&\ \ perm(c, a')\ \&\ sorted(a') \Rightarrow$
$\quad perm(a, a')\ \&\ sorted(a');$
$sorted(a, m)\ \&\ m<n\ \&\ perm(a, c)\ \&\ sorted(c, m+1)\ \Rightarrow\ sorted(c, V)\ \&\ h(c, V)<h(a, m);$
$sorted(a, m)\ \&\ m<n \Rightarrow m < n\ \&\ sorted(a, m)$

Третья формула очевидно истинна. Первая формула доказывается применением леммы pe2. Унификация термов sorted(c, m+1) и sorted(c, V) во второй формуле определяет значение переменной V: V = m+1. Остается найти h, удовлетворяющее h(c, m+1)<h(a, m). Подходящей является функция меры h(m) = n – m. Подсистема синтеза должна уметь находить такую формулу и проводить унификацию термов. В итоге получим следующую программу sort1.

```
sort1(Arn a, natn m: Arn a')
pre sorted(a, m) post perm(a, a') & sorted(a') measure n - m
{  if (m = n) a' = a
   else   { pop_into(a, m, a[m+1]: Arn c); sort1(c, m+1: a') }
};
```

Простейший способ реализации программы pop_into – последовательный обмен элемента e с предыдущими соседними элементами пока не достигнем отсортированного состояния. Это простой способ, но не эффективный: при обмене двух элементов запись элемента e в массив оказывается лишней. Предпочтительней вместо обмена элементов переместить очередной соседний элемент массива на одну позицию вверх. Схема такого алгоритма показана на рис.1.

Рис.1. Схема работы алгоритма pop_into.

На очередном цикле работы pop_into позиция k массива a является пустой. Элементы от a[k+1] до a[m+1] получены в результате смещения на одну позицию элементов a[k],…,a[m+1] исходного массива a. Все смещенные элементы больше e. Если обнаружится, что a[k-1] ≤ e, то элемент e можно будет записать в позицию k. Спецификация программы представлена модулем POP_INTO1. Предварительно следует расширить теорию Sort.

```
theory Sort { ……..
   formula sorted(Arn a, natn k, m) = ∀ i, j = k..m. i < j ⇒  a[i] ≤ a[j];
};

module POP_INTO1 {
   import Sort, Perm;
   pop_into(Arn a, natn k, m, T e: Arn a') // k – пустая позиция
   pre m < n & k > 0 & sorted(a, k-1) & sorted(a, k+1, m+1) &
            (k > m  or e < a[k+1] & a[k-1] <= a[k+1])
   post perm(a with (k: e), a') & sorted(a', m+1);
end POP_INTO;
```

Отметим, что при k = m+1 последовательность a[k+1],…,a[m+1] оказывается пустой, что учтено в спецификации альтернативой k > m.

Данная версия программы pop_into является более общей по сравнению с представленной в модуле POP_INTO. Далее можно было бы реализовать предыдущую

версию pop_into через новую. Применим другое решение – заменим вызов pop_into в программе sort1 на вызов новой версии:

```
sort1(Arn a, natn m: Arn a')
pre sorted(a, m) post perm(a, a') & sorted(a') measure n - m
{  if (m = n) a' = a
   else { pop_into(a, m+1, m, a[m+1]: Arn c); sort1(c, m+1: a') }
};
```

После такого изменения автоматически включается независимый процесс дедуктивной верификации оператора после **else**. Далее рассмотрим процесс синтеза отмеченных трех фрагментов программы pop_into.

```
pop_into(Arn a, natn k, m, T e: Arn a')
{ if (a[k-1] <= e) [*]
  else { Arn b = a with (k: a[k-1]);
          if (k = 1) [*]
          else pop_into([*])
         }
}
```

Применяя правило **QC** для внешнего условного оператора, получаем две цели:

```
[*] corr [Ppop(a, m, k, e) & a[k-1] <= e, Qpop(a, m, k, e, a') ]
{ b = a with (k: a[k-1]); if } corr [Ppop(a, m, k, e) & a[k-1] > e, Qpop(a, m, k, e, a') ]
```

Здесь Ppop и Qpop – предусловие и постусловие pop_into. Для первой цели формула корректности (1) переписывается следующим образом:

Ppop(a, m, k, e) & a[k-1] <= e & Y(a, m, k, e: a') ⇒ Qpop(a, m, k, e, a')

Здесь Y обозначает искомый оператор. После раскрытия предусловия и постусловия получаем формулу:

```
m < n & k > 0 & sorted(a, k-1) & sorted(a, k+1, m+1) &
(k > m or e < a[k+1] & a[k-1] <= a[k+1]) & a[k-1] <= e & Y(a, m, k, e: a') ⇒
        perm(a with (k: e), a') & sorted(a', m+1)
```

Чтобы сделать истинным конъюнкт perm(a **with** (k: e), a'), следует в соответствии с леммой pe1 использовать в качестве Y оператор a' = a **with** (k: e). Далее, необходимо обеспечить истинность конъюнкта sorted(a', m+1) или, эквивалентно, sorted(a **with** (k: e), m+1). Здесь надо рассмотреть два случая. Пусть k > m, что эквивалентно k = m+1. В этом случае в теории Sort необходима лемма:

```
so2: lemma k > 0 & sorted(a, k-1) & a[k-1] <= e ⇒ sorted(a with (k: e), k);
```

В случае k ≤ m необходима лемма вида:

so3**: lemma** m < n & k > 0 & sorted(a, k-1) & sorted(a, k+1, m+1) &
e < a[k+1] & a[k-1] <= e ⇒ sorted(a **with** (k**:** e), m+1);

Для второй цели применяется следующая модификация правила **QS**:

$$\textbf{QS'}:\ \frac{\begin{array}{l}P(x) \to \exists z\ B(x\textbf{:}\ z);\\ \forall z\ C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ B(x\textbf{:}\ z), Q(x, y)];\end{array}}{\text{Corr}(B(x\textbf{:}\ z); C(x, z\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x,y]}$$

В соответствии со второй посылкой цель преобразуется к виду:

{**if** (k = 1) [*] **else** pop_into([*]) } **corr**
[Ppop(a, m, k, e) & a[k-1] > e & b = a **with** (k**:** a[k-1]), Qpop(a, m, k, e, a')]

Применение правила **QC** для условного оператора дает новые две цели:

[*] **corr** [Ppop(a, m, k, e) & a[k-1] > e & b = a **with** (k**:** a[k-1]) & k = 1,
Qpop(a, m, k, e, a')]
pop_into([*]) **corr** [Ppop(a, m, k, e) & a[k-1] > e & b = a **with** (k**:** a[k-1]) & k ≠ 1,
Qpop(a, m, k, e, a')]

Построим условие корректности по первой цели в соответствии с формулой (1):

m < n & k > 0 & sorted(a, k-1) & sorted(a, k+1, m+1) &
(k > m **or** e < a[k+1] & a[k-1] <= a[k+1]) &
a[k-1] > e & b = a **with** (k**:** a[k-1]) & k = 1 & X(a, m, k, e**:** a') ⇒
perm(a **with** (k**:** e), a') & sorted(a', m+1)

Необходимо обеспечить истинность первого конъюнкта после импликации подбором соответствующего терма для a'. Ведем теорию Sortb для свойств массива b.

theory Sortb {
import Sort, Perm;
Arn a, b; natn k; T e;
sb1**: lemma** k > 0 & b = a **with** (k**:** a[k-1]) ⇒ perm(a **with** (k**:** e), b **with** (k-1**:** e));
};

В соответствии с леммой sb1 подходящим оператором на место X является a' = b **with** (k-1**:** e). Далее необходимо будет доказать истинность второго конъюнкта через обращение к SMT-решателю.

Для второй цели используем следующую модификацию правила **RB**:

$$\textbf{RBR}:\ \frac{\begin{array}{l}P(x) \to\ P_B(x)\ \&\ P^*_C(B(x));\\ P(x)\ \&\ Q_C(B(x), y) \to Q(x, y);\end{array}}{C(B(x)\textbf{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)]}$$

Правило конкретизируется следующим образом:

RB': $\dfrac{\text{Pp3(a,k,m,e,b)} \rightarrow \text{P}_B\text{(a,k,m,e)} \;\&\; \text{Ppop(}\mathbf{R}\text{)} \;\&\; \text{h(}\mathbf{R}\text{)<h(a,k,m,e);} \quad \text{Pp3(a,k,m,e,b)} \;\&\; \text{Qpop(}\mathbf{R}\text{, a')} \rightarrow \text{Qpop(a, m, k, e, a');}}{\text{pop\_into(}\mathbf{R}\text{: a') }\mathbf{corr}\text{ [Pp3(a,m,k,e), Qpop(a, m, k, e, a')]}}$

Здесь **R** обозначает искомый терм B(a,k,m,e);

Pp3(a,k,m,e,b) = Ppop(a, m, k, e) & a[k-1] > e & b = a **with** (k**:** a[k-1]) & k ≠ 1.

Раскроем Qpop во второй посылке:

Pp3(a,k,m,e,b) & perm(**x with** (**y: z**), a') & sorted(a', m+1) ⇒
perm(a **with** (k**:** e), a') & sorted(a', m+1);

Здесь **x**, **y**, **z** обозначают искомые термы. В соответствии с леммой sb1 подстановка (**x**, **y**, **z**) = (b, k-1, e) обеспечивает истинность первого конъюнкта. Далее остается подобрать меру h. Для доказательства первой посылки необходимы следующие леммы:

sb2**: lemma** m < n & k > 0 & sorted(a, k+1, m+1) &
(k > m **or** a[k-1] <= a[k+1]) & b = a **with** (k**:** a[k-1]) ⇒ sorted(b, k, m+1)
so4**: lemma** k>0 & sorted(a, k) ⇒ sorted(a, k-1)

В итоге получим следующую программу.

```
pop_into(Arn a, natn k, m, T e: Arn a') measure k
{ if (a[k-1] <= e) a' = a with (k: e)
  else { Arn b = a with (k: a[k-1]);
         if (k = 1) a' = b with (0: e)
         else pop_into(b, k-1, m, e: a')
       }
}
```

Недостаток алгоритма: если a[m] <= a[m+1] при вызове pop_into, то элемент a[m+1] сначала считывается в переменную e, а затем записывается назад в массив a. Можно было бы предварительно проверять условие a[m] > e, но это будет уже другой более сложный алгоритм. Новую программу pop_into нельзя получить простой модификацией предыдущей, поскольку меняется ее спецификация. Здесь возникает вопрос, в какой степени можно использовать процесс синтеза и верификации старой версии программы при разработке ее новой версии.

Применение оптимизирующих трансформаций для полной программы сортировки дает следующую программу на императивном расширении языка **P**:

```
sort(Arn a)
{  for (natn m = 0; ! m = n; m = m+1) {
       T e = a[m+1];
       for (natn k = m+1; ; k = k-1)
            if (a[k-1] <= e) { a[k]= e; break }
            else { a[k] = a[k-1]; if (k = 1) { a[0] = e; break }}
   }
}
```

Заключение

Программный синтез программы сортировки использует 8 лемм, которые также должны быть доказаны. При наличии этих лемм задача автоматического синтеза отмеченных операторов программы pop_into является вполне реалистичной. Без этих лемм доказать истинность генерируемых формул корректности с помощью SMT-решателей в принципе невозможно. Написать заранее нужный набор лемм более чем проблематично, особенно таких, как sb1 и sb2. Таким образом, задача автоматического синтеза реальных программ оказывается принципиально нереализуемой. Другие методы синтеза также используют леммы. И вряд ли появится чудесный метод, автоматически синтезирующий программу без лемм.

Анализ особенностей синтеза операторов в программе сортировки дает возможность сформулировать предложение об архитектуре подсистемы верификации и синтеза в системе предикатного программирования. Необходим собственный специализированный решатель, работающий интерактивно в контексте создаваемой программы и набора теорий. Решатель проводит преобразования и удобную визуализацию генерируемых формул корректности. Преобразования решателя: унификация термов, перебор термов, подстановки, обеспечивающие истинность формул с использованием лемм, и другие.

Если снабжать формулы корректности необходимыми леммами, то число формул, которые могут быть доказанными SMT-решателями, возрастает с 20% до 90%. Это и другие свойства специализированного решателя способны примерно вдвое ускорить процесс дедуктивной верификации предикатных программ. Сейчас

трудоемкость дедуктивной верификации примерно в 10 раз больше в сравнении с обычным стилем программирования. Отметим, что дедуктивная верификация самой быстрой программы сортировки [9], состоящей всего из 20 строк на императивном расширении языка **P**, потребовала более месяца.

Работа выполнена при поддержке РФФИ, грант № 16-01-00498.

Литература

1. *Карнаухов Н.С., Першин Д.Ю., Шелехов В.И.* Язык предикатного программирования P. Новосибирск, 2010. 42с. (Препр. / ИСИ СО РАН; N 153). http://persons.iis.nsk.su/files/persons/pages/plang12.pdf
2. *Manna Z., Waldinger R.* A deductive approach to program synthesis // ACM Transactions on Programming Languages and Systems, 2. 1980. P.90–121.
3. Alur R., Bod'ik R., Juniwal G., et al. Syntax-guided synthesis // FMCAD, IEEE, 2013. P. 1-8.
4. *Loding C., Madhusudan P., Neider D.* Abstract Learning Frameworks for Synthesis // TACAS 2016, LNCS 9636. 2016. P. 167-185.
5. *Kneuss E., Kuncak V., Kuraj I., Suter P.* On Integrating Deductive Synthesis and Verification Systems // EPFL Report 186043, 2013.
6. *Jacobs S., Kuncak V., and Suter Ph.* Reductions for synthesis procedures // Verification, Model Checking, and Abstract Interpretation. LNCS 7337. 2013. P. 88-107.
7. *Чушкин М.С., Шелехов В.И.* Методы синтеза фрагментов предикатных программ // Конф. «Компьютерная безопасность и криптография» SIBECRIPT'16 / Прикладная дискретная математика. Приложение. 2016. №.9. С.126-128.
8. *Чушкин М.С.* Система дедуктивной верификации предикатных программ // «Программная инженерия». № 5. 2016. С. 202-210. http://persons.iis.nsk.su/files/persons/pages/paper.pdf
9. *Шелехов В.И.* Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. N 164). http://www.iis.nsk.su/files/preprints/164.pdf
10. *Dramnesc I., Jebelean T.* Synthesis of list algorithms by mechanical proving // Jornal of Symbolic Computation, v.68, 2015. P.61-92.
11. *Шелехов В.И.* Классификация программ, ориентированная на технологию программирования // «Программная инженерия». Том 7. № 12. 2016. С. 531–538.
12. *Шелехов В.И.* Семантика языка предикатного программирования // ЗОНТ-15. Новосибирск, 2015. 13с. http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf
13. PVS Specification and Verification System. SRI International. http://pvs.csl.sri.com/
14. *Шелехов В.И.* Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия. 2011. № 2. С. 14-21.
15. Шелехов В.И. Основы предикатного программирования. — ИСИ СО РАН, Новосибирск, 2016. — 25с. http://persons.iis.nsk.su/files/persons/pages/predbase.pdf
16. http://www.iis.nsk.su/persons/vshel/files/rules.zip

Шелехов Владимир Иванович, к.т.н.,
зав.лаб. Системного Программирования Института Систем Информатики СО РАН,
Новосибирск, 630090, пр. Лаврентьева, 6
м.т. 8-905-950-82-98
vshel@iis.nsk.su

Synthesis of statements in a predicate program

Vladimir I. Shelekhov

A.P. Ershov Institute of Informatics Systems, Novosibirsk, 630090, Russian Federation,
e-mail: vshel@iis.nsk.su

Program synthesis of the marked statements in a predicate program, integrated with deductive verification and the usual style of program writing in the Eclipse editor is considered. Methods of synthesizing the statements are analyzed on the example of the effective bubble sort program. The framework for a tool integrating deductive verification and program synthesis is proposed.

Keywords: formal operational semantics, program synthesis, deductive verification, SMT solver, prototype verification system PVS, sorting algorithm.