УДК 004.43

ДЕДУКТИВНАЯ ВЕРИФИКАЦИЯ И РЕАЛИЗАЦИЯ ПРЕДИКАТНОЙ ПРОГРАММЫ ИНВЕРТИРОВАНИЯ СПИСКОВ

Шелехов Владимир Иванович

К.т.н., зав. лабораторией «Системное программирование»,
Институт систем информатики им. А.П. Ершова СО РАН,
630090, г. Новосибирск, пр. Лаврентьева 6, e-mail: vshel@iis.nsk.su

Аннотация. Представлен метод предикатного программирования в применении к известной программе инвертирования односвязных списков. Данная программа признана крайне трудной для дедуктивной верификации (verification challenge). Описываются построение и дедуктивная верификация предикатной программы инвертирования списка как объекта алгебраического типа. Эффективная императивная программа получена применением оптимизирующих трансформаций. Дедуктивная верификация предикатной программы на порядок проще верификации аналогичной императивной программы, использующей указатели.

Ключевые слова: дедуктивная верификация, трансформации программ, алгебраические типы данных, односвязный список.

Цитирование: Шелехов В.И. Дедуктивная верификация и реализация предикатной программы инвертирования списков // Информационные и математические технологии в науке и управлении. 2018. № 3 (11). С. 136–146. DOI:10.25729/2413-0133-2018-3-15

Введение. Оперирование указателями является весьма сложной и опасной процедурой в императивном программировании. Показателем такой сложности является чрезвычайная трудность дедуктивной верификации программ, оперирующих указателями.

В языке предикатного программирования Р [6] нет указателей, серьезно усложняющих программу. Вместо указателей используются объекты алгебраических типов: списки и деревья. Предикатная программа существенно проще в сравнении с императивной программой, реализующей тот же алгоритм. Эффективность предикатных программ достигается применением *оптимизирующих трансформаций*. Они определяют оптимизацию среднего уровня с переводом предикатной программы в эффективную императивную программу. Эта оптимизация отлична от классической. Базисными трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной [4, 5, 7];
- замена хвостовой рекурсии циклом;
- открытая подстановка программы на место ее вызова;
- кодирование объектов алгебраических типов (списков и деревьев) при помощи массивов и указателей.

Эффективность программы также обеспечивается оптимизацией, реализуемой программистом, на уровне предикатной программы. Итоговая программа по эффективности

не уступает написанной вручную и, как правило, короче [2, 10-12, 26]. Отметим, что в функциональном программировании (при общеизвестной ориентации на предельную компактность и декларативность [17]) оптимизация программы полностью возлагается на транслятор, в частности, обеспечивается автоматическое приведение рекурсии к хвостовому виду.

Технология предикатного программирования иллюстрируется здесь на примере известной программы инвертирования списка. Данная программа, написанная на императивном языке с использованием указателей, признана как крайне трудная для дедуктивной верификации (verification challenge). Предикатная программа инвертирования списка проста. Дедуктивная верификация этой программы на порядок проще в сравнении с верификацией соответствующей императивной программы.

В первом разделе краткое описание языка предикатного программирования. Списки определены как частный случай алгебраических типов. Во втором разделе описывается построение программы инвертирования односвязных списков. В третьем разделе описывается трансформации предикатной программы. Дедуктивной верификация программы представлена в четвертом разделе. Далее обзор работ и заключение.

1. Язык предикатного программирования. *Полная предикатная программа* состоит из набора рекурсивных *предикатных программ* на языке P [6] следующего вида:

<имя программы>(<описания аргументов>**:** <описания результатов>)
pre <предусловие>
post <постусловие>
measure <выражение>
{ <оператор> }

Необязательные конструкции предусловия и постусловия являются формулами на языке исчисления предикатов; они используются для дедуктивной верификации [10, 12, 26]. Мера используется при верификации и задается только для рекурсивных программ. Отметим, что в качестве формул могут использоваться условные логические выражения вида:

(C)? T **:** E ,

где C, T и E – формулы.

Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

<переменная> = <выражение>
{<оператор1>; <оператор2>}
<оператор1> || <оператор2>}
if (<логическое выражение>) <оператор1> **else** <оператор2>
<имя программы>(<список аргументов>**:** <список результатов>)
<тип> <пробел> <список имен переменных>

Всякая переменная характеризуется *типом* – множеством допустимых значений. *Описание типа* **type** T(p) = D с возможными параметрами p связывает имя типа T с его изображением D. Типы **bool**, **int**, **real** и **char** являются *примитивными*. Значением типа **struct**(T_1 f_1, ..., T_n f_n) является *структура* из n значений, именуемых *полями* f_1, f_2,..., f_n и имеющих типы T_1, T_2,..., T_n, соответственно. Алгебраический тип *список* определяет последовательность элементов некоторого произвольного типа T, являющегося параметром. Описание типа списка следующее:

type list (**type** T) = **union**(nil, cons(T car, list(T) cdr));

Тип list имеет два конструктора: nil, обозначающий пустой список, и cons, определяющий список, первый элемент которого представлен полем car, а остальная часть списка («хвост» – все элементы, кроме первого) определяется полем cdr. Тип list считается определенным в языке P и не требует описания в программе.

Пусть s – переменная типа список. Тогда s.car определяет первый элемент – «голову» списка s, s.cdr – список без первого элемента – «хвост» списка s. Обработка списков реализуется оператором вида:

if (s = nil) <оператор1> **else** <оператор2>; (1)

причем в <операторе2> разрешается использовать s.car и s.cdr.

Для списков s и u определены операции: len(s) – длина списка, s + u – конкатенация списков s и u. В качестве операндов конкатенации допускаются также элементы списка.

При трансляции предикатной программы в императивный язык основным способом представления списка является массив. Другими возможными альтернативами кодирования списка являются: односвязный список, двунаправленный список, кольцевой список. В языке P введены дополнительные конструкции, обеспечивающие эффективную реализацию списков через массивы [13].

2. Предикатная программа инвертирования списка.

2.1. Постановка задачи. Имеется односвязный список S. Тип списка LIST определяется описанием:

type LIST = **struct** (T car, LIST *cdr);

Здесь T – некоторый произвольный тип элемента списка, поле car определяет начальный элемент списка, поле cdr – указатель на следующий элемент. Нулевой указатель null в поле cdr означает, что данный элемент – последний в списке. Таким образом, односвязный список это набор объектов – структур типа LIST, провязанных указателями по полю cdr. Подразумевается, что цепочка элементов, определяемая указателями, не замыкается ни на одном из предыдущих элементов списка.

В задаче требуется инвертировать список S, т.е. изменить порядок элементов на обратный. Пусть список представлен последовательностью элементов:

$$S_1 \to S_2 \to \ldots \to S_n$$

Тогда результатом инвертирования будет последовательность:

$$S_1 \leftarrow S_2 \leftarrow \ldots \leftarrow S_n$$

В программе инвертирования не допускается создания новых элементов списка. Все изменения должны быть проведены в исходном списке: элементы списка (поля car) должны остаться в тех же структурных объектах, меняется лишь провязка списка, т.е. поля cdr.

Программа нетривиальна. Она иногда используется в качестве тестовой для поступающих на работу программистов.

2.2. Построение предикатной программы ReverseIn(s**:** s′) инвертирования списка s. Результат s′ – инвертированный список. Штрих в имени s′ означает, что в итоговой императивной программе s′ заменяется на s. Введем краткое имя listT для типа списка:

type T;
type listT = list(T);

Инвертирование списка s можно определить рекурсивной функцией:

formula reverse(listT s**:** listT) = (s = nil)? s **:** reverse(s.cdr) + s.car;

Спецификация программы определяется следующим образом.

ReverseIn(listT s**:** listT s′) **post** s′ = reverse(s);

Предусловие отсутствует: допускаются произвольные списки.

Применяется метод *обобщения исходной задачи* ReverseIn. Рассмотрим задачу ReverseG, в которой исходный список составлен из двух частей: s и u. Аргумент s есть начальная часть списка в инвертированном виде, аргумент u – оставшаяся часть исходного списка. Допустим, исходный список, аргумент программы ReverseIn, определяет последовательность элементов a_1, a_2, ..., a_n. Допустим, аргумент s программы ReverseG содержит k элементов. Тогда в программе ReverseG аргумент s представляет последовательность a_k, ..., a_2, a_1; аргумент u определяет оставшуюся часть списка – последовательность a_{k+1}, a_{k+2}, ..., a_n. Предположим, что последовательность a_1, a_2, ..., a_n будет кодироваться в императивной программе цепочкой:

$$A_1 \to A_2 \to ... \to A_n$$

Аргументы программы ReverseG будут представлены следующими двумя цепочками:

$$A_1 \leftarrow A_2 \leftarrow ... \leftarrow A_k \qquad\qquad A_{k+1} \to A_{k+2} \to ... \to A_n$$

Для построения постусловия программы ReverseG определим функцию reverseG, реализующую требуемое инвертирование исходного списка:

formula reverseG(listT s, u**:** listT) = reverse(u) + s;

Спецификация программы ReverseG определяется следующим образом.

ReverseG(listT s, u**:** listT s′) **post** s′ = reverseG(s, u);

Сведение исходной задачи к ReverseG реализуется следующей программой.

```
ReverseIn(listT s: listT s′) post s′ = reverse(s)
    { reverseG(nil, s: s′) };
```

Программа ReverseG представлена ниже. Она строится разложением на компоненты списка u в форме условного оператора (1).

```
ReverseG(listT s, u: listT s′) post s′ = reverseG(s, u) measure len(u) {
    if (u = nil) s′ = s
    else ReverseG(u.car + s, u.cdr: s′)
};
```

Мера len(u) используется для доказательства завершения программы.

Итак, полная программа состоит из программ ReverseIn и ReverseG. Можно ускорить данную программу, если подставить программу ReverseG на место вызова в ReverseIn.

```
ReverseIn(listT s: listT s′) post s′ = reverse(s)
    { if (s = nil) s′ = s
      else reverseG([s.car], s.cdr: s′)
    };
```

3. Трансформации. Определим набор трансформаций, превращающих программу инвертирования списков, состоящую из программ reverseIn и reverseG, в эффективную императивную программу. Первой трансформацией является склеивание $s \leftarrow s'$, заменяющее всюду переменную s′ на s. Далее проводится замена хвостовой рекурсии циклом.

Применяются упрощения. Программа reverseG подставляется на место вызова в reverseIn. Получим следующую программу:

```
reverseIn(listT s){                                                    (2)
    if (s = nil) return;
    listT u = s.cdr; s = s.car;
    while (u != nil) { s = u.car + s;  u = u.cdr }
};
```

Далее списки языка P необходимо заменить односвязными списками. Тип односвязного списка (см. разд. 2.1) определяется описанием:

```
type LIST = struct (T car, LIST *cdr);
```

Списки s и u далее будут заменены на односвязные списки S и U. В программе односвязный список представляется указателем на структуру типа LIST. Операции со списками языка **P** необходимо представить эквивалентными операциями для односвязных списков. Ниже приведено представление операций и операторов программы reverseIn.

```
s = nil      →   S = null;
u = s.cdr    →   U = S->cdr;
s = s.car    →   S->cdr = null;
u != nil     → U != null;
u = u.cdr    →  U = U->cdr
```

Кодирование оператора s = u.car + s нетривиально. Возможно следующее решение:

```
s = u.car + s → U->cdr = S;  S = U
```

Недостаток решения в том, что модифицируется (портится) U. Такой способ можно было бы использовать, если бы далее не использовалось U. Однако U используется в следующем операторе u = u.cdr. Поэтому операторы s = u.car + s и u = u.cdr необходимо кодировать совместно, используя при этом вспомогательную переменную Z для запоминания исходного значения U. Нужная трансформация приведена ниже.

```
s = u.car + s; u = u.cdr → LIST* Z = U; U = U->cdr; Z->cdr = S; S = Z
```

Отметим, что здесь операторы переставлены местами, поскольку Z->cdr = S портит U.

Итоговая программа представлена ниже.

```
reverseIn(LIST * S) {
    if (S = null) return;
    LIST* U = S->cdr; S->cdr = null;
    while (U != null) { LIST* Z = U; U = U->cdr; Z->cdr = S; S = Z }
};
```

При кодировании операций со списками используется потоковый анализ программы [4], определяющий время жизни переменных: для каждого вхождения переменной определяется, будет значение этой переменной использоваться при дальнейшем исполнении программы или нет. На основании этих данных определяются эффективные способы кодирования операций алгебраических типов.

4. Дедуктивная верификация. Предикатная программа относится к классу *программ-функций* [14], всегда нормально завершающихся с получением результата. *Спецификацией* предикатной программы H(x: y) являются два предиката: *предусловие* P(x) и *постусловие* Q(x, y). Доказана теорема, что $\mathcal{R}(H) = H$ [9], где $\mathcal{R}$ – формальная операционная

семантика [9]. *Тотальная корректность* программы относительно спецификации определяется формулой: $\forall x.\ P(x) \Rightarrow [\forall y.\ H(x{:}\ y) \Rightarrow Q(x, y)\ ]\ \&\ \exists y.\ H(x{:}\ y)$.

Для базисных операторов (параллельного, условного и суперпозиции) разработана универсальная 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. Корректность правил доказана [3] в системе PVS. В системе предикатного программирования реализован генератор формул корректности программы. Часть формул доказывается автоматически SMT-решателем CVC4. Оставшаяся часть формул генерируется для системы интерактивного доказательства PVS [24]. Данный метод опробован для дедуктивной верификации более чем 50 небольших программ [10, 12, 15, 16, 26].

В процессе дедуктивной верификации предикатной программы инвертирования списков сгенерированы 4 формулы корректности в виде следующих лемм:

lemma RB3: s' = reverseG(nil, s) $\Rightarrow$ s′ = reverse(s);
lemma COR: u = nil & s′ = s $\Rightarrow$ s′ = reverseG(s, u);
lemma r2**:** reverse(nil) = nil;
lemma r3: $\forall$listT s. $\forall$T x. reverse(s) + x = reverse(x + s)

При их доказательстве используется следующий набор библиотечных лемм:

lemma con3: u != nil $\Rightarrow$ u = u.car + u.cdr;
lemma con4: $\forall$list s. s + nil = s;
lemma con5**:** $\forall$list s. nil + s = s;
lemma con7**:** $\forall$list s, u , v. u + s = v + s $\Rightarrow$ u = v;
lemma len7**:** $\forall$list s. s != nil $\Rightarrow$ len(s.cdr) < len(s);

5. Обзор. Дедуктивная верификация программы инвертирования односвязных списков является весьма трудной. Она объявлена как сверхсложная проблема (verification grand challenge). Причина такой сложности кроется в сложности спецификации и верификации операций с указателями. Классический метод дедуктивной верификации Хоара [18] не применим при наличии в программе указателей. Большинство методов дедуктивной верификации программ с указателями базируются на логике разделения (separation logic) [25]. В каждой конференции по формальным методам встречаются работы по дедуктивной верификации с указателями, например, в последней конференции NASA Formal Methods 2017 три таких работы. Число работ с каждым годом растет; из отечественных следует отметить работу [22].

Пока ни одну из попыток проведения дедуктивной верификации односвязных списков [23, 20] нельзя назвать успешной. Спецификации длинные и сложные. Дедуктивная верификация сложна и трудоемка. Наш алгоритм инвертирования списков прост. Трудоемкость дедуктивной верификации на порядок ниже. Используемые при доказательстве леммы тривиальны.

Примечательно, что в работе [19] используется та же самая спецификация на списках и аналогичная идея обобщения задачи, что и у нас для программы reverseG. В целях улучшения понимания императивных программ с указателями используется некоторая абстрактная модель односвязных списков и логика разделения.

Заключение. Спецификация предикатной программы инвертирования списков компактна, существенно короче и проще спецификаций императивных программ, представленных в публикациях [20, 23]. Трудоемкость дедуктивной верификации предикатной программы на порядок ниже дедуктивной верификации соответствующих императивных программ для существующих ныне методов.

В языке предикатного программирования P нет указателей. Вместо указателей используются объекты алгебраических типов: списки, деревья и другие рекурсивные структуры. Для АВЛ-деревьев разработан метод эффективной трансформации операций с деревьями [1]. С этой целью язык P расширен средствами модификации поддерева, доступного по некоторому произвольному пути в дереве. Реализована эффективная нерекурсивная предикатная программа для вставки в АВЛ-дерево, аналога чему нет в функциональных языках.

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

СПИСОК ЛИТЕРАТУРЫ

1. Булгаков К.В., Каблуков И.В., Тумуров Э.Г., Шелехов В.И. Оптимизирующие трансформации списков и деревьев в системе предикатного программирования // Системная информатика, ИСИ СО РАН, Новосибирск. Электрон. журн. 2017. № 9. С. 63–92. Режим доступа: http://www.system-informatics.ru/files/article/105.pdf (дата обращения 18.04.2018)
2. Вшивков В.А., Маркелова Т.В., Шелехов В.И. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ. 2008. Т. 4 (33). С. 79–94.
3. Доказательство правил корректности операторов предикатной программы. Режим доступа: http://www.iis.nsk.su/persons/vshel/files/rules.zip (дата обращения 18.04.2018)
4. Каблуков И.В., Шелехов В.И. Реализация оптимизирующих трансформаций в системе предикатного программирования // Системная информатика, № 11. Новосибирск, 2017. С. 21–48. Электрон. журн. 2018. Режим доступа: http://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf (дата обращения 18.04.2018)
5. Каблуков И. В., Шелехов В.И. Реализация склеивания переменных в предикатной программе. Новосибирск. 2012. 6с. (Препр. / ИСИ СО РАН; N 167).
6. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.12. Новосибирск, 2013. 28с., Режим доступа: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf (дата обращения 18.04.2018).
7. Петров Э.Ю. Склеивание переменных в предикатной программе // Методы предикатного программирования. Новосибирск. ИСИ СО РАН. 2003. С. 48–61.
8. Чушкин М.С. Система дедуктивной верификации предикатных программ // Программная инженерия. 2016. № 5. С. 202–210. Режим доступа: http://persons.iis.nsk.su/files/persons/pages/paper.pdf (дата обращения 18.04.2018).
9. Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. Новосибирск. 2015. 13с. Режим доступа: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf (дата обращения 18.04.2018).

10. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия. 2011. № 2. С. 14–21.
11. Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. Новосибирск, 2004. 52с. (Препр. / ИСИ СО РАН; N 115).
12. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск. 2012. 30с. (Препр. / ИСИ СО РАН. № 164).
13. Шелехов В.И. Списки и строки в предикатном программировании // Системная информатика, ИСИ СО РАН, Новосибирск. Электрон. журн. 2014, №3. С. 25–43. Режим доступа: http://persons.iis.nsk.su/files/persons/pages/String.pdf (дата обращения 18.04.2018)
14. Шелехов В.И. Классификация программ, ориентированная на технологию программирования // Программная инженерия. Том 7. № 12. 2016. С. 531–538. Режим доступа: http://persons.iis.nsk.su/files/persons/pages/prog.pdf (дата обращения 18.04.2018)
15. Шелехов В.И. Синтез операторов предикатной программы // Труды конф. «Языки программирования и компиляторы ’2017». Ростов-на-Дону. 2017. С. 258–262. Режим доступа: http://persons.iis.nsk.su/files/persons/pages/sintr.pdf (дата обращения 18.04.2018)
16. Шелехов В.И. Доказательное построение, верификация и синтез предикатных программ // Знания-Онтологии-Теории (ЗОНТ-2017). Том 2. Институт Математики СО РАН. Новосибирск. 2017. С. 156–165. Режим доступа: http://persons.iis.nsk.su/files/persons/pages/lbase.pdf (дата обращения 18.04.2018)
17. Cooke D.E., Rushton J.N. Taking Parnas's Principles to the Next Level: Declarative Language Design. *Computer*. 2009. vol. 42. no. 9. Pp. 56–63.
18. Hoare C.A.R. An axiomatic basis for computer programming // *Communications of the ACM*. Vol. 12 (10). 1969. Pp. 576–585.
19. Jones C.B., Yatapanage N. Reasoning about Separation Using Abstraction and Reification // SEFM 2015. LNCS 9276. 2015. Pp. 3–19.
20. Leino K.R.M. Specification and verification of object-oriented software // Marktoberdorf International Summer School. 2008. 36p.
21. Leino K.R.M., Wustholz V. The Dafny Integrated Development Environment // EPTCS 149. 2014. Pp. 3–15.
22. Mandrykin M.U., Khoroshilov A.V. Region analysis for deductive verification of C programs // Programming and Computer Software. 2016. V. 42. Issue 5. Pp. 257–278.
23. Meyer B. Towards a Calculus of Object Programs // Patterns, Programming and Everything, Judith Bishop Festschrift, eds. Karin Breitman and Nigel Horspool. Springer-Verlag. 2012. Pp. 91–128.
24. PVS Specification and Verification System. SRI International. Available at: http://pvs.csl.sri.com/ (accessed 18.04.2018)
25. Reynolds J.C. Separation Logic: A Logic for Shared Mutable Data Structures // LICS '02 Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. Pp. 55–74.

26. Shelekhov V.I. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. 2011. Vol. 45. No. 7. Pp. 421–427.
27. Tagore A., Zaccai D, Weide B.W. Automatically Proving Thousands of Verification Conditions Using an SMT Solver: An Empirical Study // NASA Formal Methods. 2012. LNCS 7226. Pp. 195–209.

UDK 004.43

DEDUCTIVE VERIFICATION OF THE PREDICATE PROGRAM OF REVERSING A LINKED LIST

Vladimir I. Shelekhov
Dr., Head. Laboratory "System Programming"
A.P. Ershov Institute of Informatics Systems
Siberian Branch of the Russian Academy of Sciences
6, Acad. Lavrentjev pr., Novosibirsk 630090, Russia, e-mail: vshel@iis.nsk.su

Abstract. The simple predicate program of reversing a list as datatype object is presented. Deductive verification of this program is simple. The imperative program of reversing a linked list is obtained by the set of program transformations for the predicate program.
Keywords: program transformation, data type, linked list.

References

1. Bulgakov K.V., Kablukov I.V., Tumurov EH.G., Shelekhov V.I. Optimiziruyushchie transformacii spiskov i derev'ev v sisteme predikatnogo programmirovaniya [Optimizing transformations for operations on lists and trees in the predicate programming system] // Sistemnaya in-formatika = System Informatics, ISI SO RAN, Novosibirsk. Electronic journal. 2017. № 9. Pp. 63–92. Available at: http://www.system-informatics.ru/files/article/105.pdf, accessed 18.04.2018. (in Russian)
2. Vshivkov V.A., Markelova T.V., Shelekhov V.I. Ob algoritmah sortirovki v metode chastic v yachejkah [Sorting algorithms in the cell-participle method] // Nauchnyj vestnik NGTU = Scientific heraud of NSTU. 2008. T. 4 (33). Pp. 79–94. (in Russian)
3. Dokazatel'stvo pravil korrektnosti operatorov predikatnoj programmy [Proof of the rules of the total correctness proof for the P language]. Available at: http://www.iis.nsk.su/persons/vshel/files/rules.zip, accessed 18.04.2018. (in Russian)
4. Kablukov I.V., Shelekhov V.I. Realizaciya optimiziruyushchih transformacij v sisteme predikatnogo programmirovaniya [Optimizing transformations in the predicate programming system] // Sistemnaya informatika = System Informatics. № 11. Novosibirsk, 2017. Pp. 21–48. Electronic journal. 2018. Available at: http://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf, accessed 18.04.2018. (in Russian)

5. Kablukov I.V., Shelekhov V.I. Realizaciya skleivaniya peremennyh v predikatnoj pro-gramme [Development of the variable merging in a predicate program]. Novosibirsk. 2012. 6p. (Prepr. / ISI SO RAN. N 167). (in Russian)
6. Karnauhov N.S., Pershin D.YU., Shelekhov V.I. YAzyk predikatnogo programmirovaniya P. Versiya 0.12. [Predicative programming language P. Version 0.12.]. Novosibirsk, 2013. 28p. Available at: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf, accessed 18.04.2018. (in Russian)
7. Petrov EH. YU. Skleivanie peremennyh v predikatnoj programme [The variable merging in a predicate program] // Metody predikatnogo programmirovaniya = Predicate engineering methods. ISI SO RAN, Novosibirsk, 2003. Pp. 48-61. (in Russian)
8. Hushkin M.S. Sistema deduktivnoj verifikacii predikatnyh programm [System for deductive verification of predicate programs] // Programmnaya inzheneriya = Software Engineering. 2016. № 5. Pp. 202–210. Available at: http://persons.iis.nsk.su/files/persons/pages/paper.pdf, accessed 18.04.2018. (in Russian)
9. Shelekhov V.I. Semantika yazyka predikatnogo programmirovaniya [Semantics of the predicate programming language] // ZONT-15. Novosibirsk, 2015. 13p. Available at: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf, accessed 18.04.2018. (in Russian)
10. Shelekhov V.I. Verifikaciya i sintez ehffektivnyh programm standartnyh funkcij v tekhnologii predikatnogo programmirovaniya [Verification and synthesis of effective standard function in predicate engineering] // Programmnaya inzheneriya = Software Engineering. 2011. № 2. Pp. 14–21. (in Russian)
11. Shelekhov V.I. Razrabotka programmy postroeniya dereva suffiksov v tekhnologii predi-katnogo programmirovaniya [Tree suffix construction program in the predicate engineering]. Novosibirsk. 2004. 52 p. (Prepr. / ISI SO RAN; N 115). (in Russian)
12. Shelekhov V.I. Razrabotka i verifikaciya algoritmov piramidal'noj sortirovki v tekhnologii predikatnogo programmirovaniya [Development and verification of heap sort predicate prorams]. Novosibirsk. 2012. 30 p. (Prepr. / ISI SO RAN. № 164). (in Russian)
13. Shelekhov V.I. Spiski i stroki v predikatnom programmirovanii [Lists and strings in predicate programming] // Sistemnaya informatika = System Informatics, ISI SO RAN, Novosibirsk. Electronic journal. 2014. No. 3. Pp. 25–43. Available at: http://persons.iis.nsk.su/files/persons/ pages/String.pdf, accessed 18.04.2018. (in Russian)
14. Shelekhov V.I. Klassifikaciya programm, orientirovannaya na tekhnologiyu programmi-rovaniya [Program Classification in Software Engineering] // Programmnaya inzheneriya = Software Engineering. Tom 7. no. 12. 2016. Pp. 531–538. Available at: http://persons.iis.nsk.su/files/persons/pages/prog.pdf, accessed 18.04.2018. (in Russian)
15. Shelekhov V.I. Sintez operatorov predikatnoj programmy [Synthesis of statements in a predicate program] // Conf. «YAzyki programmirovaniya i kompilyatory ’2017 = Programming languages and compilers 2017». Rostov-na-Donu. 2017 Pp. 258–262. Available at: http://persons.iis.nsk.su/files/persons/pages/sintr.pdf, accessed 18.04.2018. (in Russian)
16. Shelekhov V.I. Dokazatel'noe postroenie, verifikaciya i sintez predikatnyh programm [Proved development, verification, and synthesis of predicate programs] // Znaniya-Ontologii-Teorii = Knoledge – Ontology - Theory (ZONT-2017). Tom 2. Institut Matematiki SO RAN.

Novosibirsk. 2017. Pp. 156–165. Available at: http://persons.iis.nsk.su/files/persons/pages/lbase.pdf, accessed 18.04.2018. (in Russian)

17. Cooke D.E., Rushton J.N. Taking Parnas's Principles to the Next Level: Declarative Language Design. *Computer*. 2009. vol. 42. no. 9. Pp. 56–63.
18. Hoare C.A.R. An axiomatic basis for computer programming // *Communications of the ACM*. Vol. 12 (10). 1969. Pp. 576–585.
19. Jones C.B., Yatapanage N. Reasoning about Separation Using Abstraction and Reification // SEFM 2015. LNCS 9276. 2015. Pp. 3–19.
20. Leino K.R.M. Specification and verification of object-oriented software // Marktoberdorf International Summer School. 2008. 36p.
21. Leino K.R.M., Wustholz V. The Dafny Integrated Development Environment // EPTCS 149. 2014. Pp. 3–15.
22. Mandrykin M.U., Khoroshilov A.V. Region analysis for deductive verification of C programs // Programming and Computer Software. 2016. V. 42. Issue 5, Pp. 257–278.
23. Meyer B. Towards a Calculus of Object Programs // Patterns, Programming and Everything, Judith Bishop Festschrift, eds. Karin Breitman and Nigel Horspool. Springer-Verlag. 2012. Pp. 91–128.
24. PVS Specification and Verification System. SRI International. Available at: http://pvs.csl.sri.com/ (accessed 18.04.2018)
25. Reynolds J.C. Separation Logic: A Logic for Shared Mutable Data Structures // LICS '02 Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. Pp. 55–74.
26. Shelekhov V.I. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. 2011. Vol. 45. No. 7. Pp. 421–427.
27. Tagore A., Zaccai D, Weide B.W. Automatically Proving Thousands of Verification Conditions Using an SMT Solver: An Empirical Study // NASA Formal Methods. 2012. LNCS 7226. Pp. 195–209.