Автоматизация доказательства условий корректности программ и перспективы применения машинного обучения в данной области

Кондратьев Дмитрий Александрович

Институт систем информатики им. А. П. Ершова СО РАН

Дедуктивная верификация

Дедуктивная верификация позволяет свести задачу проверки корректности программ к задаче доказательства специальных формул

Входные данные:

Программа, аннотированная спецификациями:

- Предусловие
- Постусловие
- Инварианты циклов

Неформальная постановка задачи:

Доказать, что "программа делает то, что записано в ее спецификациях".

Логика Хоара

Формула в логике Хоара

Тройка Хоара: $\{P\}\ S\ \{Q\}$

- P — предусловие (логическая формула)
- S — программа
- Q — постусловие (логическая формула)

Истинность тройки Хоара

Частичная корректность тройки Хоара $\{P\}\ S\ \{Q\}$ означает, что "если предусловие P истинно перед исполнением фрагмента программы S, и, если исполнение S завершилось, тогда постусловие Q выполняется после его завершения".

Тотальная корректность тройки Хоара $[P]\ S\ [Q]$ означает, что "если предусловие P истинно перед исполнением фрагмента программы S, тогда исполнение S завершается и постусловие Q выполняется после его завершения".

Свойства частичной и тотальной корректности

Пусть

- P' – свойство "предусловие P истинно перед исполнением фрагмента программы S";
- T' – свойство "исполнение фрагмента программы S завершается";
- Q' – свойство "постусловие Q выполняется после завершения фрагмента программы S".

Тогда

- $(P' \wedge T') \rightarrow Q'$ – метазапись свойства частичной корректности;
- $P' \rightarrow (T' \wedge Q')$ – метазапись свойства тотальной корректности.

Способ доказательства тотальной корректности

Для доказательства тотальной корректности можно доказать отдельно два теоремы:

- $(P' \wedge T') \to Q'$ – теорема о частичной корректности;
- $P' \to T'$ – теорема о завершимости.

Из двух данных теорем следует истинность свойства тотальной корректности:

$$P' \to (T' \wedge Q')$$

Как доказать теорему о завершимости?

Метод Флойда для доказательства завершимости программы

Цикл – единственная инструкция программы, исполнение которой может не завершиться.

Метод Флойда: программа завершается, если каждому циклу удается сопоставить меру.

Мера – это функция, удовлетворяющая следующим свойствам:

- Областью значений меры является фундированное множество (частично упорядоченное множество, у которого каждое непустое подмножество имеет минимальный элемент).
- Значение меры строго убывает после каждой итерации цикла.

Пример доказательства завершимости

Цикл, возводящий элементы массива в квадрат:

```
for (i = 0; i < n; i++) {
   a[i] = a[i]*a[i]}
```

Функция меры:

$$M(i, n) = n - i$$

- Множество значений $M(n, i)$ – множество натуральных чисел (фундированное множество).
- Значение функции $M(n, i)$ строго убывает после каждой итерации цикла.

Доказательство частичной корректности

Схема правила вывода:

$$\frac{\psi_1, \ldots, \psi_n}{\varphi}$$

- ψ_1,...,ψ_n – посылки правила вывода (набор троек Хоара и логических формул)
- φ — заключение правила вывода (тройка Хоара)

Логическая система, содержащая аксиомы и правила вывода для всех синтаксических форм языка программирования, называется логикой Хоара или аксиоматической семантикой языка

Правило вывода для последовательного исполнения

$$\frac{\{P\}\ \mathsf{prog}_1\ \{R\},\ \{R\}\ \mathsf{prog}_2\ \{Q\}}{\{P\}\ \mathsf{prog}_1;\ \mathsf{prog}_2\ \{Q\}}$$

Правила вывода условий корректности

Правило вывода для присваивания переменной:

$$\frac{\{P\}\ \text{prog};\ \{Q(var \leftarrow expr)\}}{\{P\}\ \text{prog};\ \text{var=expr}\ \{Q\}}$$

Правило вывода для присваивания элементу массива:

$$\frac{\{P\}\ \text{prog};\ \{Q(a \leftarrow upd(a,i,expr))\}}{\{P\}\ \text{prog};\ \text{a[i]=expr}\ \{Q\}}$$

Правило вывода для *if*:

$$\frac{\{P \wedge B\}\ \text{S}_1;\ \text{prog}\ \{Q\},\ \{P \wedge \neg B\}\ \text{S}_2;\ \text{prog}\ \{Q\}}{\{P\}\ \text{if B then S}_1\ \text{else S}_2;\ \text{prog}\ \{\text{Q}\}}$$

Правило вывода для пустой программы:

$$\frac{P \rightarrow Q}{\{P\}\text{emptyProgram}\{Q\}}$$

Проблема инвариантов циклов

Правило вывода для цикла *while*

$$\frac{\{P\}\ \text{prog};\ \{I\},\quad \{I \wedge B\}\ \text{S}\ \{I\},\quad I \wedge \neg B \rightarrow Q}{\{P\}\ \text{prog};\ \text{while B inv I do S}\ \{Q\}}$$

где I — инвариант цикла

Инвариант цикла – это утверждение, которое истинно перед исполнением цикла и для каждой итерации цикла и обеспечивает корректность на выходе из цикла.

Первый пример

Соревнование по верификации программ "Java program verification challenge"

`negate_first` — задача из данного соревнования

Реализация программы `negate_first` на языке C:

```
void negate_first(int n, int* a) {
    int i;
    for (i = 0; i < n; i++) {
        if (a[i] < 0) {a[i] = -a[i]; break;}}}
```

Данная программа изменяет знак первого отрицательного элемента массива

Данная программа содержит следующие конструкции, вызывающие сложности при дедуктивной верификации:

1. Возможное присваивание элементу массива в цикле
2. Возможное исполнение инструкции `break` в цикле

Первый пример

`negate_first`: предусловие:

$$(a_0 = a) \wedge (0 < n) \wedge (n \leq \mathit{length}(a_0))$$

`negate_first`: постусловие:

$$\begin{gathered}(\neg \mathit{found\_negative}(n,\ a_0) \rightarrow \\ a = a_0) \\ \wedge \\ (\mathit{found\_negative}(n,\ a_0) \rightarrow \\ a = \mathit{update}(a_0, \\ \mathit{count\_index}(n,\ a_0), -a_0[\mathit{count\_index}(n,\ a_0)]))\end{gathered}$$

Предикат *found_negative* истинен тогда и только тогда, когда в массиве есть отрицательный элемент
Функция *count-index* считает индекс первого отрицательного элемента в случае его наличия в массиве

Первый пример

`negate_first`: цикл:

```
    for (i = 0; i < n; i++) {
        if (a[i] < 0) {a[i] = -a[i]; break;}}}
```

`negate_first`: инвариант цикла:

$$(0 < n) \wedge (n \leq \mathit{length}(a_0)) \wedge (0 \leq i) \wedge (i \leq n) \wedge (0 \leq j) \wedge (j < i)$$
$$\rightarrow$$
$$(0 \leq a[j] \wedge (a[j] = a_0[j])$$

Первый пример

`negate_first`: первое условие корректности:

$$((a_0 = a) \wedge (0 < n) \wedge (n \leq \mathit{length}(a_0)))$$
$$\rightarrow$$
$$((0 < n) \wedge (n \leq \mathit{length}(a_0)) \wedge (0 \leq 0) \wedge (0 \leq n) \wedge (0 \leq j) \wedge (j < 0)$$
$$\rightarrow$$
$$(0 \leq a[j]) \wedge (a[j] = a_0[j])$$

Первый пример

`negate_first`: второе условие корректности:

$$
\begin{gathered}
(((0 < n) \wedge (n \leq \mathit{length}(a_0)) \wedge (0 \leq i) \wedge (i \leq n) \wedge (0 \leq j) \wedge (j < i) \\
\rightarrow \\
(0 \leq a[j]) \wedge (a[j] = a_0[j])) \\
\wedge \\
\neg(i < n)) \\
\rightarrow \\
((\neg \mathit{found\_negative}(n,\ a_0) \rightarrow \\
a = a_0) \\
\wedge \\
(\mathit{found\_negative}(n,\ a_0) \rightarrow \\
a = \mathit{update}(a_0, \\
\mathit{count\_index}(n,\ a_0), -a_0[\mathit{count\_index}(n,\ a_0)])))
\end{gathered}
$$

Первый пример

`negate_first`: третье условие корректности:

$$
\begin{gathered}
(((0 < n) \wedge (n \leq \mathit{length}(a_0)) \wedge (0 \leq i) \wedge (i \leq n) \wedge (0 \leq j) \wedge (j < i) \\
\rightarrow \\
(0 \leq a[j]) \wedge (a[j] = a_0[j])) \\
\wedge \\
(i < n) \\
\wedge \\
\neg(a[i] < 0)) \\
\rightarrow \\
((0 < n) \wedge (n \leq \mathit{length}(a_0)) \wedge (0 \leq i + 1) \wedge (i + 1 \leq n) \wedge \\
(0 \leq j) \wedge (j < i + 1) \\
\rightarrow \\
(0 \leq a[j]) \wedge (a[j] = a_0[j]))
\end{gathered}
$$

Первый пример

`negate_first`: четвертое условие корректности:

$$
\begin{gathered}
(((0 < n) \wedge (n \leq \mathit{length}(a_0)) \wedge (0 \leq i) \wedge (i \leq n) \wedge (0 \leq j) \wedge (j < i) \\
\rightarrow \\
(0 \leq a[j]) \wedge (a[j] = a_0[j])) \\
\wedge \\
(i < n) \\
\wedge \\
(a[i] < 0)) \\
\rightarrow \\
((\neg \mathit{found\_negative}(n,\ a_0) \rightarrow \\
a = a_0) \\
\wedge \\
(\mathit{found\_negative}(n,\ a_0) \rightarrow \\
a = \mathit{update}(a_0, \\
\mathit{count\_index}(n,\ a_0), \\
-a_0[\mathit{count\_index}(n,\ a_0)])))
\end{gathered}
$$

Второй пример

Сортировка простыми вставками

```
 1. /* P */
 2. void insertion_sort(int a[], int n){
 3.    int k, i, j;
 4.    for (i = 1; i < n; i++){
 5.       k = a[i];
 6.       for(j=i-1;j>=0;j--){
 7.          if (a[j] <= k) break;
 8.          a[j + 1] = a[j];}
 9.       a[j + 1] = k;}}
10. /* Q */
```

Предусловие P: $0 < n \wedge a = a_0 \wedge n \leq \mathit{length}(a_0)$

Постусловие Q: $\mathit{perm}(0, n-1, a_0, a) \wedge \mathit{ord}(0, n-1, a)$

perm — предикат перестановочности третьего и четвертого аргумента в диапазоне от первого до второго аргумента
ord — предикат упорядоченности

Второй пример

Циклы из примера:

```
4.    for (i = 1; i < n; i++){
5.       k = a[i];
6.       for(j=i-1;j>=0;j--){
7.          if (a[j] <= k) break;
8.          a[j + 1] = a[j];}
9.       a[j + 1] = k;}
```

Инвариант внешнего цикла:

$$i \leq n \wedge n \leq length(a) \wedge length(a_0) = lenght(a) \wedge$$
$$a_0[i : n-1] = a[i : n-1] \wedge perm(0, i-1, a_0, a) \wedge ord(0, i-1, a)$$

Инвариант внутреннего цикла (*less*: аргумент $\leq$ элементы массива):

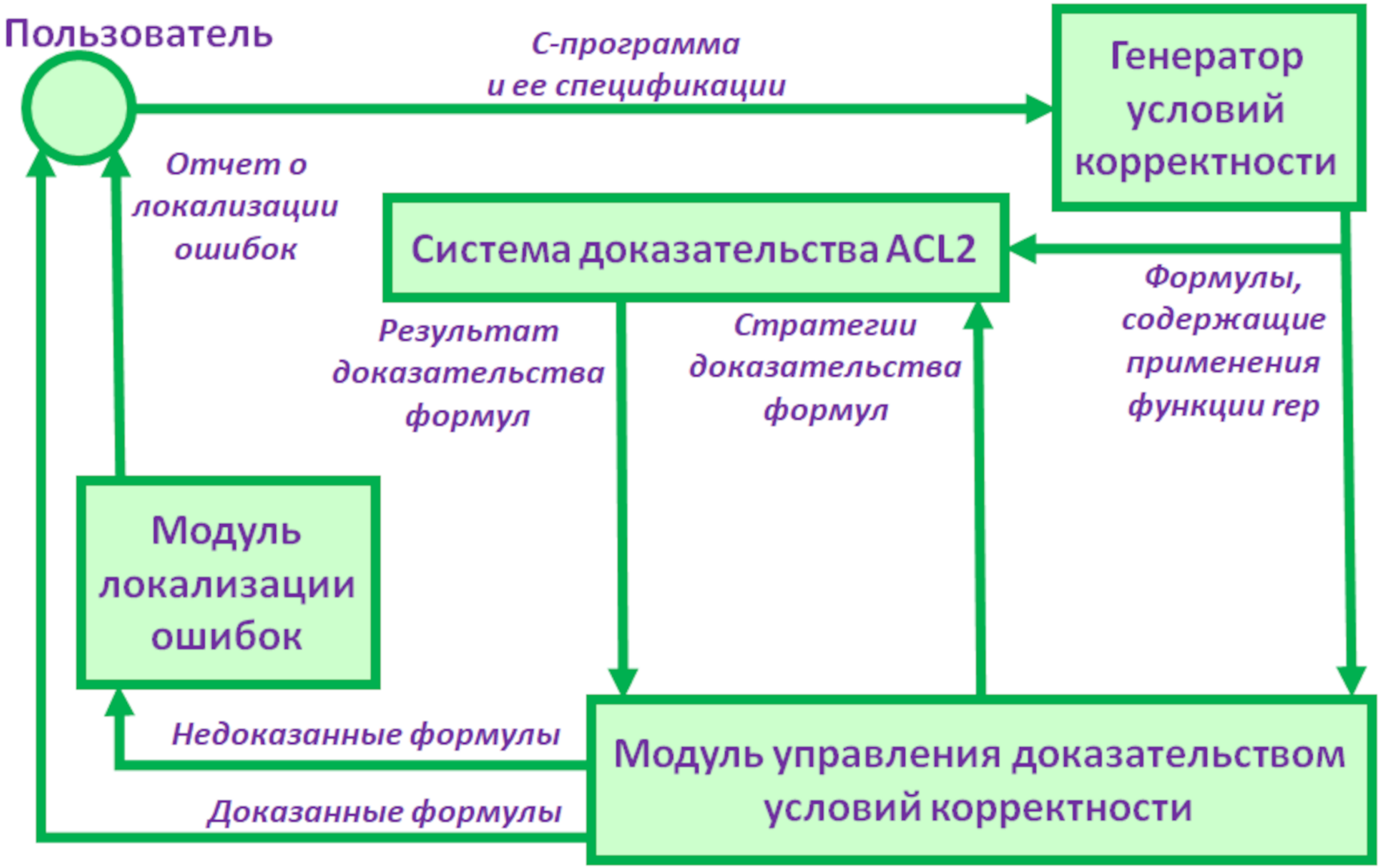
$$1 \leq i \leq n \wedge n \leq length(a) \wedge length(a_0) = lenght(a) \wedge$$
$$a_0[i : n-1] = a[i : n-1] \wedge 0 \leq j \leq i-1 \wedge ord(0, j, a) \wedge ord(j+1, i, a) \wedge$$
$$less(k, a, j+1, i) \wedge perm(0, i, a_0, update(a, j+1, k)) \wedge$$
$$((0 \leq j < i-1) \rightarrow (a[j] \leq a[j+2]))$$

Система C-lightVer

C-lightVer — система дедуктивной верификации C-программ

- **Вход** – программа на языке C и ее формальные спецификации (заданные в виде логических формул)
- **Первый этап исполнения** – генерация условий корректности для входной программы и ее спецификаций
- **Второй этап исполнения** – доказательство условий корректности (используется автоматизированная система доказательства ACL2)
- **Третий этап исполнения** – применение методов локализации ошибок в случае наличия недоказанных условий корректности
- **Выход** – отчет о результатах верификации (если все условия корректности доказаны, то программа соответствует спецификациям).

Система C-lightVer: обзор



Задача автоматизации дедуктивной верификации:

1. **Проблема:** задание инвариантов циклов
   Предлагаемое нами решение: символический метод верификации финитных итераций
2. **Проблема:** автоматизация доказательства условий корректности
   Предлагаемое нами решение: стратегии доказательства условий корректности

Символический метод верификации финитных итераций

Решение проблемы задания инвариантов для циклов специального вида

Рассмотрим финитную итерацию

$$\texttt{for } x \texttt{ in } S \texttt{ do } v := body(v, x) \texttt{ end}$$

- v является кортежом переменных цикла
- $empty(S)$ – проверка структуры S на пустоту
- $choo(S)$ – операция выборки элемента из структуры S
- $rest(S)$ – структура S за исключением $choo(S)$

Определим рекурсивную функцию rep**, заменяющую финитную итерацию**

Пусть v_0 обозначает начальные значения переменных из v.

- Если $empty(S)$, тогда $rep(v_0, S, body) = v_0$,
- Если $\neg empty(S)$, тогда $rep(v_0, S, body) = body(rep(v_0, rest(S), body), choo(S))$.

Правило вывода для финитных итераций

Правило вывода для финитных итераций без инвариантов

$$\frac{\{P\}\,\mathsf{pr};\{Q(v \leftarrow \mathit{rep}(v, S, \mathit{body}, n))\}}{\{P\}\,\mathsf{pr};\ \mathsf{for}\,(\mathsf{i} = 0; \mathsf{i} < \mathsf{n}; \mathsf{i}++)\ \mathsf{v} := \mathsf{body}(\mathsf{v}, \mathsf{i})\ \mathsf{end}\{Q\}}$$

Система доказательства ACL2

Логика, основанная на вычислимых рекурсивных функциях

Applicative Common Lisp (ACL) – язык системы ACL2. *S*-выражения языка Lisp. Префиксная запись: сначала операция, потом операнды.

Логические операции: *and*, *or*, *not*, *implies*

Задание формул: конструкции *defun* и *define*

Все рекурсивные функции должны быть завершимыми

ACL2 может автоматически находить меру для доказательства завершимости функции

Меру для доказательства завершимости функции можно задать вручную с помощью конструкции : *measure*

Моделирование типов в языке Applicative Common Lisp

Типы моделируются с помощью двух функций:

- Предикат, проверяющий принадлежность типу. Обычно название таких предикатов – это название типа с добавлением *p* в конце. Например,
  - *natp* – проверка принадлежности к натуральным числам,
  - *integerp* – проверка принадлежности к целым числам,
  - *integer-listp* – проверка принадлежности к спискам целых чисел.

  В системе ACL2 заданы аксиомы для таких встроенных предикатов.
- Функции приведения к значению типа. Обычно название таких предикатов – это сокращенное название типа с добавлением *fix* в конце. Например,
  - *nfix* – приведение к натуральным числам,
  - *ifix* – приведение к целым числам,
  - *integer-list-fix* – приведение к спискам целых чисел.

  В системе ACL2 заданы аксиомы для таких встроенных функций.

Моделирование типов в языке Applicative Common Lisp

Типичный код на языке ACL2 – проверка принадлежности к типу с помощью импликации, например:

$$(implies\ (integerp\ i)\ (equal\ (+\ i\ 1)\ (+\ i\ 1)))$$

Альтернатива – использовать явное приведение к типу:

$$(equal\ (+\ (ifix\ i)\ 1)\ (+\ (ifix\ i)\ 1))$$

Две важные операции над выражениями типа список: функции *nth* и *update-nth*.

Если i — индекс, l — список, то $(nth\ i\ l)$ — значение i-го элемента списка l.

Если *expr* — выражение языка ACL2, то *(update-nth i expr l)* — новый список, который совпадает со списком l за исключением i-го элемента, значением которого является значение *expr*.

Пример определения функции на языке Applicative Common Lisp

Вычисление суммы абсолютных значений элементов списка в диапазоне от i до j:

```
(defun abs-sum(i j a)
    (if (or (not (natp i)) (not (natp j))) 0
        (if (> i j) 0
            (if (= i j) (abs (nth j a))
                (+ (abs (nth j a))
                   (abs-sum i (- j 1) a))))))
```

Конструкция $b*$

Макрос $b*$ позволяет промоделировать последовательное исполнение инструкций. Он является расширением макроса $let*$ языка ACL2, позволяющего удобным образом задать вложенный let. Рассмотрим общий вид конструкции $b*$:

$$(b* \ (\ldots(var \ expr)\ldots) \ result)$$

где $(var \ expr)$ означает связывание var со значением $expr$, которое может зависеть от связанных ранее переменных.

Значением $b*$ является $result$

Библиотека *fty*

Моделировать новые типы данных позволяет библиотека *fty* языка ACL2

Макрос *fty::defprod* задает тип, аналогичный инструкции struct языка C.

Если *st* — такой тип, то с помощью *fty::defprod* будет сгенерирован конструктор, макросы *make-st, change-st* и функции доступа к значениям полей.

Пусть *fd* — поле структуры *s*, имеющей тип *st*. Тогда *(change-st s :fd expr)* — новая структура типа *st*, совпадающая со структурой *s* за исключением поля *fd*, значением которого является значение *expr*.

(st->fd s) значение поля *fd* структуры *s*. Другим способом доступа к данному значению является *s.fd*.

Алгоритм генерации *rep*: трансляция на язык ACL2

Алгоритм транслирует конструкции тела цикла на язык ACL2.

Генерируется тип структуры *frame* с помощью *defprod*. Поля такой структуры соответствуют изменяемым переменным цикла. Функция *rep* возвращает объект типа *frame*.

Все объекты типа *frame* называются *fr*. Каждая инструкция цикла моделируется как создание нового объекта *fr*, поля которого являются обновлением полей прежнего *fr*.

Последовательное исполнение моделируется с помощью $b*$:

$$(b* \ (\ldots(var \ expr)\ldots) \ result)$$

(*var expr*) означает связывание *var* со значением *expr*, которое может зависеть от связанных ранее переменных. В качестве *var* используется *fr*, в качестве *expr* – обновление полей *fr*.

Для моделирования выхода из цикла мы используем булевское поле *loop-break* объекта *fr*. Это поле истинно только после срабатывания `break`.

Трансляция допустимых конструкций, функция gen_rep

- пустая инструкция *empty*, *gen_rep*(empty) = (*fr fr*)
- инструкция break;, *gen_rep*(break;) = ((*when t*) *fr*)
- присваивание c = b;, где *b* – C-kernel выражение
  gen_rep(c = b;) = (*fr* (*change-frame fr* :*c b*))
- присваивание a[i] = b;, где *b* – C-kernel выражение
  gen_rep(a[i] = b;) =
  (*fr* (*change-frame fr* :*a* (*update-nth i b fr.a*)))
- инструкция if (c) b else d, где *b* и *d* — допустимые конструкции
  gen_rep(if (c) b else d) =
  (*fr* (*if c* (*b* ∗ (*gen_rep*(*b*)) *fr*) (*b* ∗ (*gen_rep*(*d*)) *fr*)))
  ((*when fr.loop-break*) *fr*)
- блок $\{a_1\ a_2\ \ldots\ a_{k-1}\ a_k\}$, где a_r — допустимая конструкция
  gen_rep($\{a_1\ a_2\ \ldots\ a_{k-1}\ a_k\}$) =
  (*fr* (b^*(*gen_rep*(a_1) ... *gen_rep*(a_k)) *fr*))
  ((*when fr.loop-break*) *fr*)
- вложенная финитная итерация
  gen_rep(for (j = 0; j < m; j + +) u := body(u, j) end) =
  (*fr* (*rep*$_k$ *m fr*$_k$))

Первый пример

negate_first: определение структуры frame:

```
(fty::defprod frame
  ((loop-break booleanp)
   (a integer-listp)
   ))

(define frame-init
  ((a integer-listp)
   )
  :returns (fr frame-p)
  (make-frame
   :loop-break nil
   :a a
 )
  ///
  (fty::deffixequiv frame-init))
```

Первый пример: определение *rep*

```
(define rep ((i natp) (fr frame-p))
  :measure (nfix i)
  :guard (<= i (len (frame->a fr)))
  :verify-guards nil
  :returns (upd-fr frame-p)
  (b* (((when (zp i)) (frame-fix fr))
       (fr (rep (- i 1) fr))
       ((when (frame->loop-break fr)) fr)
       (fr (if
             (< (nth (- i 1) (frame->a fr)) 0)
             (b*
                 ((fr (change-frame fr :a
                                    (update-nth (- i 1)
                                      (- (nth (- i 1) (frame->a fr)))
                                      (frame->a fr))))
                  (fr (change-frame fr :loop-break t)))
               fr)
             fr))
       ((when (frame->loop-break fr)) fr))
    fr))
```

Теоремы в языке ACL2

Задание теорем: конструкции *defrule* и *defthm*

Все переменные в теоремах находятся под неявными кванторами всеобщности. Исключения: моделирование кванторов с помощью конструкций *defun* – *sk*.

Подсказки для доказательства задаются с помощью конструкции *hints*

Основная подсказка – это конструкция *use*, добавляющая определенную лемму в качестве посылки доказываемой теоремы после переименования переменных леммы.

Первый пример: условие корректности

```
(defrule my-theorem1
  (b* (((mv found-spec index-spec) (count_index n a_0))
       ((frame fr) (rep n (frame-init a))))
  (implies
   (and
    (integer-listp a)
    (integer-listp a_0)
    (equal a a_0)
    (integerp n)
    (< 0 n)
    (<= n (length a_0)))
     (and
      (implies (not found-spec)
               (equal fr.a a_0))
      (implies found-spec
               (equal
               fr.a
                (update-nth index-spec (- (nth index-spec a_0))
                 a_0)))))))
```

Второй пример

Условие корректности программы сортировки простыми вставками:

$$0 < n \wedge a = a_0 \wedge n \leq \mathit{length}(a_0) \rightarrow$$
$$\mathit{perm}(0, n-1, a_0, \mathit{rep}_1(n, a_0).a) \wedge \mathit{ord}(0, n-1, \mathit{rep}_1(n, a_0).a)$$

rep_1 – операция замены для внешнего цикла

rep_2 – операция замены для внутреннего цикла

В определении rep_1 используется rep_2

Автоматизация доказательства в системе ACL2

Два основных способа автоматизации доказательства в системе ACL2:

- Автоматические подстановки в доказываемой теореме с помощью лемм класса *rewrite*.
- Автоматический запуск доказательства по индукции, используя в качестве схем индукции определения применяемых в доказываемой теореме функций.

Схемы индукций, пример *dec-induct*

```
(defun dec-induct (n)
    (if (zp n) nil (dec-induct (- n 1))))
```

Схема:

База индукции `zp n`
Шаг индукции: `n-1 -> n`

Схемы индукции, примеры

Определение функции

```
(defun abs-sum(i j a)
   (if (or (not (natp i)) (not (natp j))) 0
       (if (> i j) 0
           (if (= i j) (abs (nth j a))
               (+ (abs (nth j a))
               (abs-sum i (- j 1) a))))))
```

Схема индукции

База индукции: i > j or i = j

Шаг индукции: i, j-1, a -> i, j, a

Леммы класса *rewrite*

Теорема класса rewrite (без конструкций для задания имени теоремы) имеет следующий вид:

$$(implies\ (and\ h_1\ \ldots\ h_n)\ (equiv\ lhs\ rhs))$$

где

1. *implies* является импликацией;
2. *and* является конъюнкцией;
3. *equiv* является отношением эквивалентности (либо равенство *equal* либо эквивалентность *iff*);
4. $(and\ h_1 \ldots h_n)$ является условием применения теоремы;
5. $h_1\ \ldots\ h_n$ являются гипотезами;
6. *lhs* является шаблоном;
7. *rhs* является замещающим выражением.

Теоремы класса *rewrite*

Выполняются два алгоритма сопоставления. Первый алгоритм состоит в сопоставлении выражения доказываемой теоремы и шаблона правила. Выражения доказываемой теоремы сопоставляются с шаблонами правил в порядке, обратном заданию теорем, соответствующих правилам. Если первый алгоритм сообщает об успешном сопоставлении, то применяется второй алгоритм.

Второй алгоритм состоит в сопоставлении посылки доказываемой теоремы и условия применения. Второй алгоритм пытается выполнить подстановку первого алгоритма в посылке доказываемой теоремы и в условии применения. Потом второй алгоритм пытается сопоставить все конъюнкты условия применения с отдельными конъюнктами доказываемой теоремы. Если сопоставление успешно, то к замещающему выражению применяется подстановка первого алгоритма, и выражение доказываемой теоремы заменяется на полученное замещающее выражение.

Стратегии доказательства условий корректности

Проблема автоматизации доказательства условий корректности

Условия корректности могут содержать применения рекурсивных функций, соответствующих циклам

Предложен набор стратегий для системы доказательства ACL2

Эти стратегии основаны на генерации лемм о свойствах программ с помощью специальных шаблонов

Леммы о рекурсивных функциях, соответствующих циклам, также генерируются

Например, может быть сгенерирована лемма о случае исполнения инструкции `break`

Эти леммы позволяют системе доказательства автоматизировать доказательство условий корректности

Первый пример

`negate_first`: два случая: иструкция break исполнилась или нет. Рассмотрим стратегию, основанную на использовании информации об исполнении break. Применение этой стратегии приводит к генерации следующей леммы:

$$
\begin{gathered}
((a_0 = a) \wedge (0 < n) \wedge (n \leq \mathit{length}(a_0))) \\
\rightarrow \\
(\mathit{rep}(n,\ a).\mathit{loop\text{-}break} = \mathit{found\_negative}(n,\ a_0) \\
\wedge \\
(\neg \mathit{found\_negative}(n,\ a_0) \rightarrow \\
\mathit{rep}(n,\ a).a = a_0) \\
\wedge \\
(\mathit{found\_negative}(n,\ a_0) \rightarrow \\
\mathit{rep}(n, a).a = \mathit{update}(a_0, \\
\mathit{count\_index}(n,\ a_0), -a_0[\mathit{count\_index}(n,\ a_0)])))
\end{gathered}
$$

`negate_first`: Система доказательства ACL2 успешно доказывает условие корректности, используя индукцию по n и эту лемму.

Второй пример: стратегия автоматизации доказательства условий корректности для программ с вложенными циклами

Рассмотрим случай, когда возрастание количества итераций внешнего цикла приводит к возрастанию части последовательности, обрабатываемой внутренним циклом. Тогда попытаемся доказать условие корректности, используя индукцию по количеству итераций внешнего цикла.

Пусть условие корректности имеет вид $P \to Q$, где

- Q — заключение условия корректности, зависящее от rep_i.
- P — посылка условия корректности.
- a — массив, над которым исполняется итерация.
- n — длина массива.
- rep_i — функция замены для i-того цикла, тело которой содержит применение rep_{i+1}.

Стратегия заключается в доказательстве такого условия корректности индукцией по n.

Второй пример: автоматическое доказательство условия корректности программы сортировки простыми вставками

Сначала применяется стратегия для программ с вложенными циклами – автоматически запускается индукция по количеству итераций внешнего цикла.

База индукции автоматически доказывается системой ACL2.

Шаг индукции автоматически доказывается системой ACL2 с помощью лемм, добавленных в теорию предметной области

- стратегией для программ, спецификации которых содержат функции со свойством конкатенации,
- стратегией для программ с финитными итерациями над массивами.

Стратегия автоматизации доказательства условий корректности для программ, спецификации которых содержат функции со свойством конкатенации

Определим такое свойство предикатов, как свойство конкатенации. Будем говорить, что предикат R обладает свойством конкатенации, если для R выполнено свойство вида

$$R(i, k, u_1, \ldots, u_n) \wedge R(k+1, j, u_1, \ldots, u_n) \rightarrow R(i, j, u_1, \ldots, u_n)$$

Например, свойству конкатенации удовлетворяет предикат перестановочности массивов. Предикат перестановочности массивов проверяет, являются ли массивы-аргументы перестановкой друг друга в заданном диапазоне индексов.

Стратегия для программ, спецификации которых содержат функции со свойством конкатенации

Будем говорить, что предикат R обладает свойством конкатенации со склейкой на границе по предикату f, если для R выполнено свойство следующего вида:

$$(R(i,k,u_1,\ldots,u_n) \wedge R(k+1,j,u_1,\ldots,u_n) \wedge \\ f(u_1[k],u_1[k+1]) \wedge \ldots f(u_m[k],u_m[k+1])) \rightarrow \\ R(i,j,u_1,\ldots,u_n)$$

Для каждого $m(1 \leq m \leq n)$ будем называть выражение $f(u_m[k],u_m[k+1])$ условием склейки на границе для свойства конкатенации.

Например, свойству конкатенации со склейкой на границе по предикату $\leq$ удовлетворяет предикат упорядоченности массива по возрастанию.

Стратегия для программ, спецификации которых содержат функции со свойством конкатенации

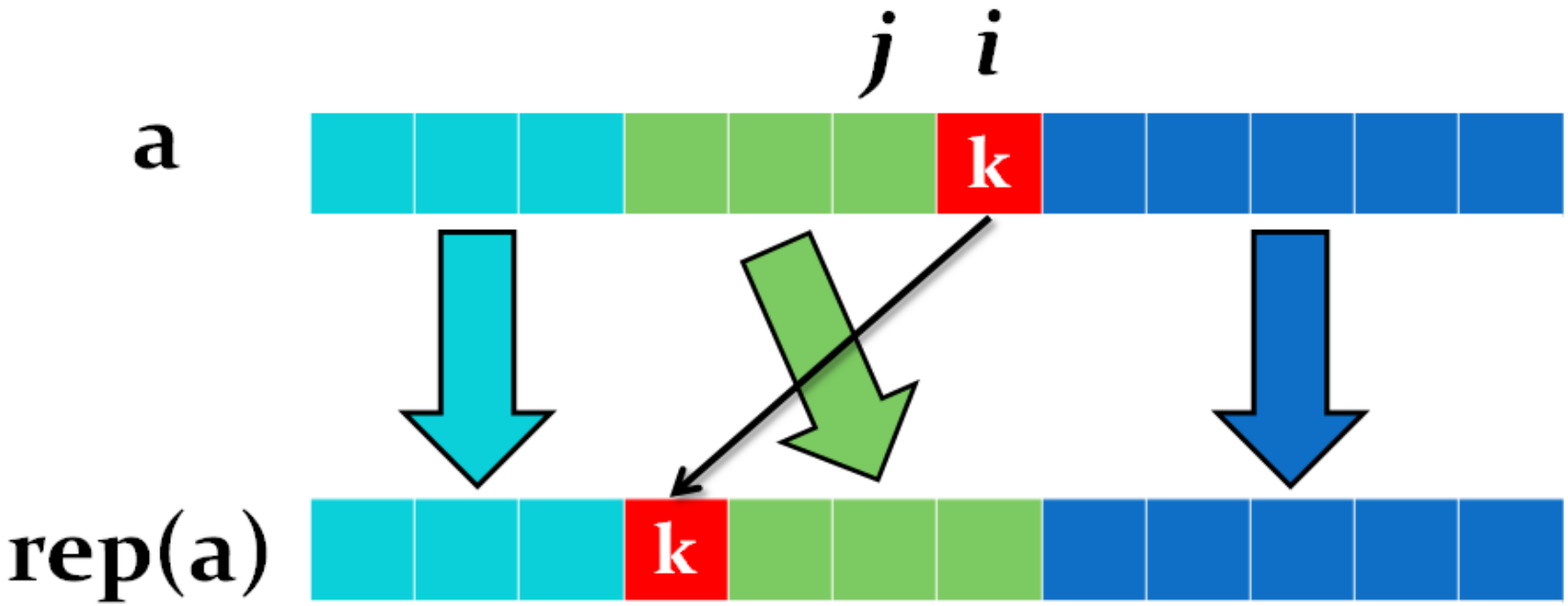
Пусть A – массив, $B = rep(A)$. Стратегия генерирует утверждения о равенстве подмассивов A и B с помощью эвристик:

- Если счетчик цикла с инструкцией `break` возрастает (убывает), то генерируется утверждение, что равны подмассивы массивов B и A, начинающиеся с итогового значения счетчика (с нуля) до длины массива минус 1 (до итогового значения счетчика).
- Если i — счетчик цикла с выражениями вида $A[i + expr] = A[i]$, то генерируется утверждение, что подмассив массива B является результатом сдвига подмассива массива A.
- Если счетчик цикла возрастает (убывает) на произвольной итерации цикла, то генерируется утверждение, что равны подмассивы массивов B и A, начинающиеся с текущего значения счетчика (с нуля) до длины массива минус 1 (до текущего значения счетчика).

Стратегия для программ, спецификации которых содержат функции со свойством конкатенации

Утверждения о равенстве подмассивов поступают на вход системе ACL2. В результате образуется множество D пар подмассивов, равенство которых удалось доказать.
Каждый предикат из постусловия проверяется на выполнение свойства конкатенации или свойства конкатенации со склейкой на границе по предикату, найденному синтаксическим анализом теории предметной области. Для каждого предиката S с такими свойствами стратегия генерирует леммы о выполнении S для вторых элементов пар D, исходя из выполнения S для первых элементов пар D. Для предиката со свойством конкатенации со склейкой на границе стратегия генерирует леммы о выполнении условия склейки на границах вторых элементов пар D. Эти леммы помогают ACL2 доказать выполнение свойства S для подмассивов, полученных в результате объединения вторых элементов пар D. Эти леммы помогают доказать свойство конкатенации для результирующего массива.

Второй пример: применение стратегии для программ, спецификации которых содержат функции со свойством конкатенации



Второй пример: стратегия автоматизации доказательства условий корректности для программ с вложенными циклами

Рассмотрим случай, когда возрастание количества итераций внешнего цикла приводит к возрастанию части последовательности, обрабатываемой внутренним циклом. Тогда попытаемся доказать условие корректности, используя индукцию по количеству итераций внешнего цикла.

Пусть условие корректности имеет вид $P \to Q$, где

- Q — заключение условия корректности, зависящее от rep_i.
- P — посылка условия корректности.
- a — массив, над которым исполняется итерация.
- n — длина массива.
- rep_i — функция замены для i-того цикла, тело которой содержит применение rep_{i+1}.

Стратегия заключается в доказательстве такого условия корректности индукцией по n.

Второй пример: стратегия автоматизации доказательства условий корректности для программ с вложенными циклами

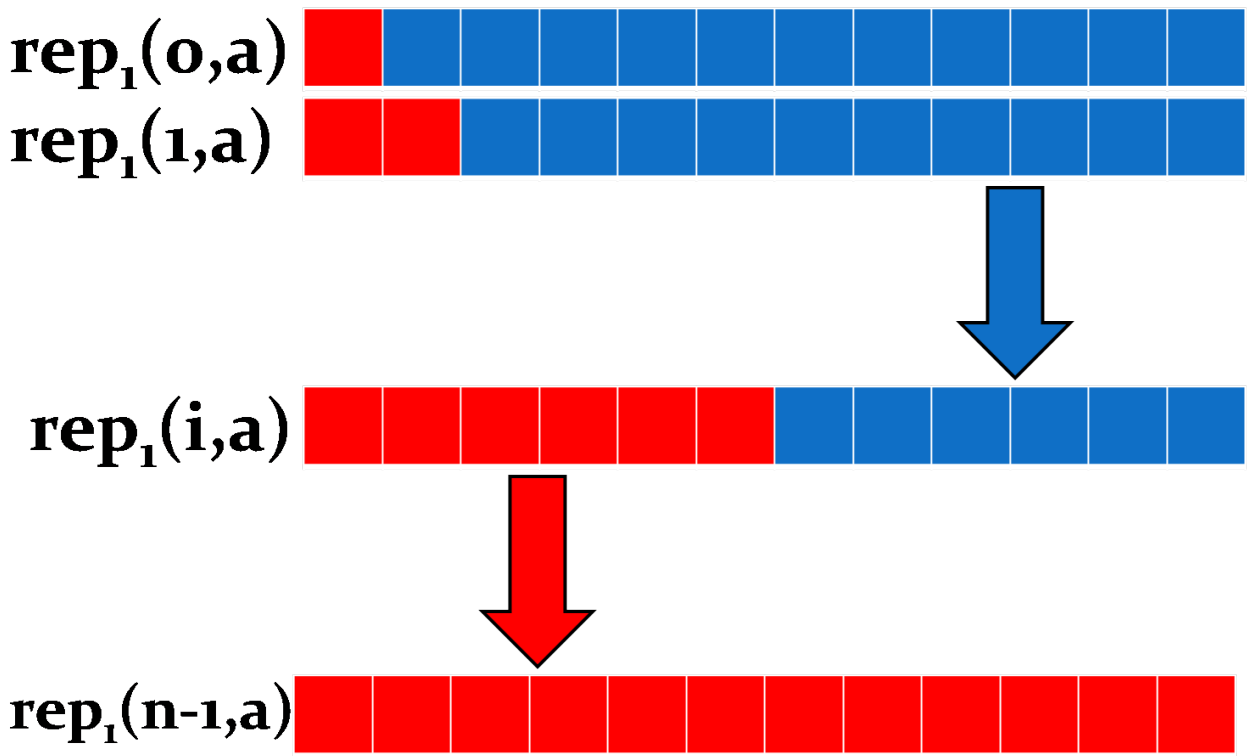
Такая стратегия может быть полезна, если внутренний цикл определяет ту часть последовательности, над которой исполняется каждая итерация внешнего цикла. Тогда подстановка вместо операции замены для внешнего цикла ее определения приводит к формуле, где доказываемое свойство сформулировано относительно внутреннего цикла. Доказывать это свойство позволяет индукционная гипотеза о части последовательности, обрабатываемой внутренним циклом.

Второй пример: стратегия для программ над изменяемыми массивами

Сгенерируем лемму:

$$
\begin{gathered}
(iteration \in N) \wedge (index \in N) \wedge \\
(iteration \leq env.upper\text{-}bound) \wedge (env.upper\text{-}bound < (lenfr.a)) \wedge \\
(fr.j = (env.upper\text{-}bound - 1)) \\
(index \neq expr\text{-}ind_1) \\
\ldots \\
(index \neq expr\text{-}ind_w) \\
\rightarrow \\
rep.a(iteration - 1, env, fr)[index] = \\
rep.a(iteration, env, fr)[index]
\end{gathered}
,
$$

Второй пример: применении стратегии для программ с финитными итерациями над массивами в случае сортировки простыми вставками



Организатор соревнования по дедуктивной верификации в рамках контеста VeHa-2023

Кондратьев Дмитрий Александрович, к.ф.-м.н., научный сотрудник Института систем информатики им. А.П. Ершова СО РАН.

Обратная связь: письма на электронную почту apple-66@mail.ru

Контест VeHa-2023

Два соревнования:

- Соревнование по дедуктивной верификации.
- Соревнование по model checking.

Тематика данных соревнований покрывает два главных направления формальной верификации программ:

- Дедуктивная верификация.
- Model checking.

Рассмотрим соревнование по дедуктивной верификации программ.

Ближайший аналог – соревнование VerifyThis

Задача по дедуктивной верификации программы, предназначенной для решения такой проблемы миссии "Луна-25", как проверка равенства всех приоритетов в массиве команд

Цель задачи состоит в том, чтобы задать формальные спецификации программы, предназначенной для решения такой проблемы миссии "Луна-25", как проверка равенства всех приоритетов в массиве команд, и провести автоматическую дедуктивную верификацию данной программы в системе C-lightVer.

Задача по дедуктивной верификации программы, предназначенной для решения такой проблемы миссии "Луна-25", как проверка равенства всех приоритетов в массиве команд

Рассматриваемая задача является первым приближением к решению проблемы миссии "Луна-25" и состоит в верификации программы, проверяющей, все ли элементы массива равны друг другу.

То есть, это абстракция к проблеме миссии "Луна-25", которая состоит в том, что в массив команд с одинаковым приоритетом попала команда, приоритет которой отличается от всех остальных.

Проверка, есть ли в массиве хотя бы один элемент, отличающийся от остальных, могла бы позволить избежать данной проблемы.

Задача по дедуктивной верификации программы, предназначенной для решения такой проблемы миссии "Луна-25", как проверка равенства всех приоритетов в массиве команд

Задача состоит в том, чтобы для программы, проверяющей, все ли элементы массива равны, задать постусловие и теорию предметной области с определением применяемой в постусловии функции.

Необходимо, чтобы заданное постусловие и теория предметной области с определением функции, применяемой в постусловии, позволили дедуктивно верифицировать данную программу в системе C-lightVer.

Обучение участников соревнования по дедуктивной верификации в рамках контеста VeHa-2023

До соревнования были подготовлены и размещены на сайте соревнования:

- Пособие для участников соревнования по дедуктивной верификации в рамках контеста VeHa-2023
- Презентация тьюториала для участников соревнования по дедуктивной верификации в рамках контеста VeHa-2023

Также был проведен тьюториал для участников соревнования по дедуктивной верификации в рамках контеста VeHa-2023. Видеозапись тьюториала была выложена на сайт соревнования VeHa-2023.

Кроме того, участникам соревнования было настоятельно рекомендовано установить систему C-lightVer до соревнования.

Исходные данные

```
/* (and (integer-listp a) (natp n) (< 0 n)
(<= n (length a))) */
int element_equality(int *a, int n){
    int result = 1;
    for (int i = 1; i < n; i++)
    {
        if (a[i-1] != a[i])
        {
            result = 0;
            break;
        }
    }
    return result;
}
/*  */
```

Классическое решение: постусловие

```
/* (and (integer-listp a) (natp n) (< 0 n)
(<= n (length a))) */
int element_equality(int *a, int n){
    int result = 1;
    for (int i = 1; i < n; i++)
    {
        if (a[i-1] != a[i])
        {
            result = 0;
            break;
        }
    }
    return result;
}
/* (= (element-equality 0 (- n 1) a) result) */
```

Классическое решение: теория предметной области

```
(in-package "ACL2")

(include-book "std/util/defrule" :dir :system)
(include-book "centaur/fty/top" :dir :system)
(include-book "std/util/bstar" :dir :system)
(include-book "std/typed-lists/top" :dir :system)
(include-book "std/lists/top" :dir :system)
(include-book "std/basic/inductions" :dir :system)
```

Классическое решение: теория предметной области

```
(defun element-equality(i j a)
    (if
        (or
            (not
                (natp
                    i
                )
            )
            (not
                (natp
                    j
                )
            )
        )
        0
```

Классическое решение: теория предметной области

```
(if
    (>=
        i
        j
    )
    1
```

Классическое решение: теория предметной области

```
(if
    (not
        (equal
            (nth
                (-
                    j
                    1
                )
                a
            )
            (nth
                j
                a
            )
        )
    )
    0
    (element-equality
        i
        (-
            j
            1
        )
        a)))))
```

Результаты

Из 15 команд, участвовавших в соревновании VeHa-2023, решения задачи по дедуктивной верификации прислали 13 команд.

Из них 10 баллов набрали 9 команд.

Из них 9 баллов набрали 3 команды.

И 8 баллов набрала 1 команда.

Основные ошибки: диапазон элементов массива в постусловии функции.

Интересные решения: Артем Кокорин

```
/* (and (integer-listp a) (natp n) (< 0 n)
(<= n (length a))) */
int element_equality(int *a, int n){
    int result = 1;
    for (int i = 1; i < n; i++)
    {
        if (a[i-1] != a[i])
        {
            result = 0;
            break;
        }
    }
    return result;
}
/* (= (element-equality n a) result) */
```

Интересные решения: Артем Кокорин

```
(defun element-equality-bool(j a)
    (if (not (natp j)) nil
        (if (= 0 j) t
          (and (= (nth (- j 1) a) (nth j a))
              (element-equality-bool (- j 1) a)
          )
        )
    )
)

(defun element-equality(n a)
  (if (element-equality-bool (- n 1) a) 1 0)
)
```

Интересные решения: команда Cache-Invalidation

```
(min
    (if
        (=
            (nth
                j
                a
            )
            (nth
                (-
                    j
                    1
                )
                a
            )
        )
        1
        0
    )
    (element-equality
        i
        (-
            j
            1
        )
        a))))))
```

Перспективы применения машинного обучения для автоматизации дедуктивной верификации

Автоматизация основных этапов дедуктивной верификации программ с помощью машинного обучения:

- Генерация формальных спецификаций программ
- Доказательство условий корректности программ
- Локализация ошибок в программах при дедуктивной верификации

Применение машинного обучения для генерации формальных спецификаций программ

Первый этап дедуктивной верификации состоит в задании следующих видов формальных спецификаций:

- Предусловие
- Инварианты циклов
- Оценочные функции
- Постусловие

Генерация формальных спецификаций программ с помощью машинного обучения.

В первую очередь генерация инвариантов циклов, так как задание инвариантов циклов является известной проблемой на пути автоматизации формальной верификации.

Генерация формальных спецификаций программ с помощью машинного обучения

Искусственный интеллект может предлагать формулы, которые являются кандидатами на роль формальных спецификаций.

Преимущество такого подхода состоит в следующем упрощении процесса задания формальных спецификаций: вместо задания спецификаций "с нуля"пользователь системы верификации будет либо выбирать сгенерированные искусственным интеллектом спецификации, либо вносить в сгенерированные искусственным интеллектом спецификации небольшие правки.

Родственные работы по проблеме задания формальных спецификаций

Kondratyev D.A., Maryasov I.V., Nepomniaschy V.A. The Automation of C Program Verification by the Symbolic Method of Loop Invariant Elimination // Automatic Control and Computer Sciences. 2019. Volume 53. Issue 7. pp. 653–662. DOI: `https://doi.org/10.3103/S0146411619070101`

В данной статье описан эвристический метод решения проблемы инвариантов циклов, который позволяет избежать задания инвариантов определенных видов циклов с помощью символической замены данных циклов рекурсивными функциями.

Но данный метод может применяться не ко всем классам циклов.

Родственные работы по проблеме задания формальных спецификаций

Li J., Sun J., Li L., Loc Le Q., Lin S-W., “Automatic Loop Invariant Generation and Refinement through Selective Sampling”, 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, 782–792.

В данной статье описана система генерации инвариантов циклов, основанная на машинном обучении.

Но система, описанная в данной статье, не поддерживает циклы, где применяются операции break и присваивание элементам массивов

Родственные работы по проблеме задания формальных спецификаций

В работе "Code2Inv: A Deep Learning Framework for Program Verification" (DOI: https://doi.org/10.1007/978-3-030-53291-8_9) описана система Code2Inv для генерации инвариантов циклов.

Данная система основана на нейронных сетях. Но система Code2Inv не поддерживает вложенные циклы.

Применение машинного обучения для генерации формальных спецификаций программ

Искусственный интеллект может генерировать и деревья применения тактик доказательства, которые являются кандидатами на роль доказательств условий корректности, и формулы, которые являются кандидатами на роль лемм, применение которых упрощает доказательство условий корректности.

Искусственный интеллект может также пытаться генерировать доказательства для кандидатов на роль лемм.

Искусственный интеллект может предлагать в качестве лемм только те формулы, доказательство для которых ему удастся сгенерировать.

Родственные работы по проблеме доказательства условий корректности

Towards Automatic Deductive Verification of C Programs over Linear Arrays // Lecture Notes in Computer Science. 2019. Volume 11964. pp. 232–242. DOI: https://doi.org/10.1007/978-3-030-37487-7_20

В данной статье описаны эвристические стратегии доказательства условий корректности, которые основаны на генерации лемм для упрощения доказательства условий корректности программ с циклами определенного вида.

Но данный метод, в отличие от предлагаемого нами подхода, может применяться не ко всем классам циклов.

Родственные работы по проблеме доказательства условий корректности

В статье "Proof-pattern recognition and lemma discovery in ACL2" (DOI: https://doi.org/10.1007/978-3-642-45221-5_27) описана основанная на машинном обучении система ACL2(ml), предназначенная для генерации лемм, применение которых упрощает доказательство целевой теоремы.

Но обученная на общих теориях из стандартной библиотеки ACL2 система ACL2(ml) плохо справляется с генерацией лемм в теориях предметных областей, используемых при дедуктивной верификации программного обеспечения.

Родственные работы по проблеме доказательства условий корректности

Sanchez-Stern A., Alhessi Y., Saul L., Lerner S. Generating correctness proofs with neural networks // Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2020). 2020. pp. 1-10. DOI: https://doi.org/10.1145/3394450.3397466

В данной статье описана система Proverbot9001 для автоматизации доказательства условий корректности компилятора CompCert, заданных в системе доказательства Coq.

Но данная система ориентирована на дедуктивную верификацию только компилятора CompCert.

Применение машинного обучения для автоматизации локализации ошибок при дедуктивной верификации

Искусственный интеллект может предлагать набор значений переменных условия корректности, который будет являться кандидатом на роль контрпримера к условию корректности, а также набор значений переменных программы, который будет являться кандидатом на роль теста, демонстрирующего ошибку.

Родственные работы по проблеме локализации ошибок

Kondratyev D.A., Nepomniaschy V.A. Automation of C Program Deductive Verification without Using Loop Invariants // Programmin and Computer Software. 2022. Volume 48. Issue 5. pp. 331–346. DOI: https://doi.org/10.1134/S036176882205005X

В данной статье описана эвристические стратегии проверки некорректности циклов над последовательностями данных с помощью доказательства таких свойств данных циклов, выполнимость которых может означать наличие ошибок в данных циклах.

Но данные стратегии применимы только к циклам над последовательностями данных.

Родственные работы по проблеме локализации ошибок

В классической статье "SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques" (DOI: https://doi.org/10.1007/3-540-45657-0_20) описан метод генерации контрпримеров с помощью SAT-решателей и машинного обучения.

Но данный метод более ориентирован на верификацию на основе проверки на модели (model checking), чем на дедуктивную верификацию программ.

Родственные работы по проблеме локализации ошибок

В статье "Constructions in combinatorics via neural networks" (DOI: https://doi.org/10.48550/arXiv.2104.14516) описан метод генерации контрпримеров для доказательства ложности формул с помощью нейронных сетей.

Но данный метод ориентирован на доказательство ложности формул о свойствах графов.

Формальная верификация искусственного интеллекта

Рассмотрим работы в области формальной верификации искусственного интеллекта, основанного на нейронных сетях. В статье "Probabilistic Verification of Neural Networks Against Group Fairness" (DOI: https://doi.org/10.1007/978-3-030-90870-6_5) описан метод формальной верификации нейронных сетей с помощью вероятностной проверки на модели (model checking). Но данный метод ориентирован на проверку выполнения только одного свойства нейронной сети о вероятности ошибки в выходных данных нейронной сети. Перспективные идеи описаны в статье "Neuro-Symbolic Verification of Deep Neural Networks" (DOI: https://doi.org/10.48550/arXiv.2203.00938). В данной статье описан язык для задания функциональных свойств нейронных сетей и дедуктивная верификация нейронных сетей с помощью нейросимволического подхода.

Литература

1. **Статья о нашей системе дедуктивной верификации C-lightVer:**
   Kondratyev D.A., Nepomniaschy V.A. Automation of C Program Deductive Verification without Using Loop Invariants // Programming and Computer Software. 2022. Volume 48. Issue 5. pp. 331–346. DOI: https://doi.org/10.1134/S036176882205005X
2. **Статья о нашем решении проблемы автоматизации дедуктивной верификации:**
   Kondratyev D. Implementing the Symbolic Method of Verification in the C-Light Project // Lecture Notes in Computer Science. 2018. Volume 10742. pp. 227–240. DOI: https://doi.org/10.1007/978-3-319-74313-4_17
3. **Статья о нашем решении проблемы инвариантов циклов:**
   Kondratyev D.A., Maryasov I.V., Nepomniaschy V.A. The Automation of C Program Verification by the Symbolic Method of Loop Invariant Elimination // Automatic Control and Computer Sciences. 2019. Volume 53. Issue 7. pp. 653–662. DOI: https://doi.org/10.3103/S0146411619070101
4. **Статья о нашем решении проблемы автоматизации доказательства условий корректности:**
   Kondratyev D., Maryasov I., Nepomniaschy V. Towards Automatic Deductive Verification of C Programs over Linear Arrays // Lecture Notes in Computer Science. 2019. Volume 11964. pp. 232–242. DOI: https://doi.org/10.1007/978-3-030-37487-7_20

Автоматизация доказательства условий корректности программ и перспективы применения машинного обучения в данной области

Кондратьев Дмитрий Александрович

Институт систем информатики им. А. П. Ершова СО РАН