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

В.А. Батраков, В.И. Шелехов

АВТОМАТИЧЕСКОЕ ДОКАЗАТЕЛЬСТВО ФОРМУЛ КОРРЕКТНОСТИ ПРЕДИКАТНОЙ ПРОГРАММЫ В СИСТЕМЕ RUSSELL

Препринт
163

Новосибирск 2012

Описывается опыт разработки транслятора с языка предикатного программирования P в язык системы автоматического доказательства Russell с целью доказательства формул корректности предикатных программ.

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

V.A. Batrakov, V.I. Shelekhov

AUTOMATIC PROVING OF CORRECTNESS FORMULAS FROM PREDICATE PROGRAMS IN THE RUSSELL SYSTEM

Preprint
163

Novosibirsk 2012

The development of a translator of the P language formulas into the language of the Russell automatic prover system with the purpose of proving the correctness formulas in Russell is described.

1. ВВЕДЕНИЕ

По мере развития информационных технологий сложность программного обеспечения (ПО) постоянно росла, что влекло за собой увеличение числа ошибок в нем и ущерба от этих ошибок. Для обеспечения надежности работы программных систем большое значение имеют различные методы верификации, позволяющие выявлять ошибки на разных этапах разработки и сопровождения ПО, чтобы последовательно устранять их. Одним из таких методов является дедуктивная верификация, реализуемая построением формул корректности – математических утверждений, истинность которых необходимо доказать чтобы гарантировать правильность программы.

Доказательство утверждений вручную подвержено ошибкам. Гарантировать безошибочное доказательство можно лишь в рамках системы автоматического доказательства на компьютере. Подобное доказательство – сложный и трудоемкий процесс. Существующие системы автоматического доказательства второго поколения, такие как **Mizar**, **Coq**, **HOL** и др., далеки от идеала. Ф. Видайк утверждает [3], что в системе **Mizar**, например, многие математические понятия просто не могут быть выражены, а система **HOL** обладает «закрытой» архитектурой, что является серьезным ограничением. Кроме того, в системе **HOL** не существует ни подтипов, ни даже зависимых типов.

В системе предикатного программирования реализуются две компоненты: генератор условий корректности предикатных программ относительно спецификаций и статический анализатор динамической семантики предикатных программ.

Генератор условий корректности строит набор формул, истинность которых гарантирует стопроцентную правильность программы относительно спецификации. Доказательство сгенерированных формул является сложной и трудоемкой задачей.

Статический анализ динамической семантики проводится в трансляторе с языка предикатного программирования **P** в соответствии с правилами семантической корректности конструкций предикатной программы. Большую часть этих правил транслятор проверяет статически. В сложных случаях транслятор генерирует условие корректности в виде логической формулы, которая должна быть доказана.

Результат работы каждой из компонент – набор теорий на языке предикатного программирования **P**. Теории содержат условия корректности пре-

дикатной программы – формулы, которые должны быть доказаны в одной из систем автоматического доказательства: PVS или Russell.

В настоящей работе описана реализация доказательства формул в системе Russell, которые предварительно должны быть транслированы на язык системы Russell. Для осуществления трансляции необходимо определить образ входящих в формулы конструкций языка предикатного программирования: каждая конструкция языка P представляется некоторым фрагментом на языке Russell, использующим дополнительные определения, помещаемые в библиотеку системы Russell. Так как язык системы Russell не имеет встроенных типов, в библиотеке также необходимо определить систему типов языка предикатного программирования.

Следующей задачей является поддержка доказательства оттранслированных формул в системе Russell. Для упрощения процесса доказательства необходимо добавлять в библиотеку вспомогательные леммы для конструкций языка P, определенных в виде образов в системе Russell.

В главе 2 приведено описание языка предикатного программирования, а в главе 3 – описание языка Russell. В главе 4 описывается образ языка P в языке Russell. В главе 5 подробнее рассматривается трансляция теорий. Глава 6 содержит примеры трансляции программ с языка P на язык Russell. В главе 7 рассматривается доказательство теорем в системе Russell. В заключении подводятся итоги работы, описываются основные проблемы.

2. ВЫРАЖЕНИЯ И ТИПЫ ЯЗЫКА ПРЕДИКАТНОГО ПРОГРАММИРОВАНИЯ

Язык предикатного программирования P [1] – это язык функционального программирования, в котором сочетаются функциональный и операторный (предикатный) стили записи алгоритмов. Он обладает существенно большей выразительностью по сравнению с чисто функциональными языками. По синтаксису, набору операторов и операций язык приближен к стилю языков семейства C. Эффективность предикатных программ достигается применением системы оптимизирующих трансформаций, переводящих программы на язык императивного расширения языка P.

Условия корректности генерируются в составе теорий, которые могут содержать описания переменных, описания типов, а также описания формул и лемм. Формулы – выражения логического типа. Леммы – утверждения на языке исчисления предикатов, содержащие выражения, операции и

типы языка P. Таким образом, необходимо транслировать весь язык P, за исключением операторов.

Язык P содержит следующие типы:

- базовые числовые типы:
  - **nat** (тип натуральных чисел);
  - **int** (тип целых чисел);
  - **real** (тип вещественных чисел);
  - **bool** (тип булевых переменных);
  - **char** (символьный тип);
- структурные типы:
  - массив – **array**(Tval, T_1 , … , T_n), где Tval – тип элементов массива, T_i – типы индексов;
  - структура – **struct** (<тип поля> F_1 , … ,<тип поля> F_n), где F_i – имена полей, значение типа структуры состоит из совокупности значений набора полей;
  - объединение – **union** (constr(<тип поля> f_1, … ,<тип поля> f_n),…) – набор конструкторов (альтернатив), содержащих поля (каждый конструктор состоит из набора полей), f_i – имя поля;
  - множество подмножеств – **set**(T), где T – тип элементов базового множества;
  - предикатный тип – **predicate** <описание спецификации предиката>;

Частными случаями типа объединения являются структуры следующего вида:

```
type list(typeT) = union {
 nil,
 cons(T car, list( T ) cdr)
};
```

- строка (**string**) – частный случай списка, эквивалентен **list** (**char**);
- перечисление – **enum**(T_1 , … , T_n), где T_i – имена полей, т.е. перечисления – частный случай объединений с конструкторами без аргументов.

Выражения языка P включают

- унарные операции (– унарный минус, ^ – возведение в степень, ! – отрицание, …);
- бинарные операции (* – умножение, / – деление, % – остаток от деления, & – логическое “и”, …);

- условное выражение (**c ? a : b);**
- кванторное выражение (<квантор> <список переменных>. <выражение>)
- элемента массива:
  <элемент массива> ::=
  <переменная типа массив>[<индекс массива>];
- агрегаты, определяющие значения структурных типов:
  агрегат ::=<изображение типа> (<элементы агрегата>);
- модификацию массивов и структур:
  <выражение> **with** <агрегат>;
- мультипеременные и мультивыражения – списки переменных и выражений, соответственно; мультивыражение может находиться в правой части оператора присваивания, если в левой находится мультипеременная:

  мультивыражение ::= список выражений | … |
  список выражений;

  мультипеременная ::= список переменных | … |
  список переменных;

3. ЯЗЫК СИСТЕМЫ RUSSELL

Система автоматического доказательства Russell названа в честь выдающегося философа и логика Бертрана Рассела. Система транслирует исходные тексты с языка Russell на язык metamath, а точнее, на его подмножество smm (Simplifed MetaMath). Язык metamath, разработанный Н. Мегилом в начале 90-х годов XX века, является очень простым, надежным и универсальным. Но metamath обладает одним очень важным недостатком – доказательства в metamath синтаксически абсолютно не ясны. Язык системы Russell же, напротив, обладает прозрачной структурой и понятен пользователю, сохраняя при этом уровень универсальности и надежности языка metamath. Библиотека **Russell**'а содержит более 12000 теорем из различных областей современной математики. В частности, она позволяет работать с вещественными, целыми, натуральными числами, функциями, множествами и различными операциями с ними.

В языке Russell можно выделить следующие основные конструкции:

Локальные синтаксические единицы, к которым можно обращаться только в локальном контексте:

- Описание переменой :
  var <имя переменной> ":" <имя типа>
- Описание терма–выражения, служащего для описания грамматики выражений в Russell:
  ":" <имя типа> "=" "#" <выражение> ";"
- Описание формулы:
  <номер формулы> ":" <имя типа> "=" "|-" <выражение> ";"
  Формулы бывают двух видов: гипотезы (hyp) и заключения (prop).

Глобальные синтаксические единицы:

- Константа (constant) – символ, входящий в выражения языка, не являющийся переменной:
  constant {
  symbol <символ> ;
  }
- Теории (theory) являются аналогом пространства имен:
  theory <имя теории>;
  contents of <имя теории> {…};

- Правило (rule) сопоставляет выражению его тип:
  rule <имя правила> (<описания переменных>) {
  term : <имя типа> = # <выражение> ;
  }

Декларация типов (**type**) позволяет объявить новый тип или подтип уже существующего (описания надтипов могут отсутствовать):

type <имя типа> : <имя надтипа>, …, <имя надтипа>;

- Аксиома (**axiom**) – недоказуемое постулируемое утверждение (здесь wff – тип формулы, hyp i – гипотезы, prop – утверждение):
  axiom <имя аксиомы> (<описания переменных>)
  {
  hyp 1 : wff = |- <выражение> ;
  …
  hyp N : wff = |- <выражение>;
  ---
  prop : wff = |- <выражение> ;
  }

- Определение (**definition**) –то же самое что и аксиомы; устанавливает соответствие между определяемым (**defiendum**) и определяющим (**definiens**) выражениями:
  definition <имя определения> (<описания переменных>)
  {
  defiendum : wff = # <выражение> ;
  definiens : wff = # <выражение> ;
  ---
  prop : wff = |- (defiendum <-> definiens) ;
  }
- Теорема (theorem) – утверждение с полным доказательством.
  theorem <имя теоремы> (<описания переменных>))
  {
  hyp 1 : wff = |- <выражение> ;
  hyp 2 : wff = |- <выражение> ;
  ---
  prop : wff = |- <выражение> ;
  }
  proof { <доказательство> }
- Проблема (problem) – утверждение без полного доказательства.

Доказательство теорем представляется как линейный вывод и состоит из последовательности шагов (step) и вложенных подутверждений (claim). Отсутствие типов в **Russell**'е (как числовых, так и алгебраических) является одной из основных проблем трансляции и должно быть восполнено построением соответствующих стандартных библиотек теорий в системе **Russell**. Библиотеки должны также содержать формулы и леммы для упрощения процесса доказательства.

4. ОБРАЗЫ ВЫРАЖЕНИЙ И ТИПОВ

Описание структуры образа произвольной конструкции языка P в соответствующую конструкцию языка Russell реализуется в следующей форме:

tr(<языковая конструкция>) = <образ конструкции>

4.1. Образы выражений

Перейдем к более детальному определению образов конструкций и выражений языка P.

- **Образы унарных операций:**
  tr (^) = ↑
  tr (-) = -u
  tr (!) = ¬
- **Образы бинарных операций:**
  tr (*) = · **tr (/) = /**
  tr (a % b) = a mod b **tr (+) = +**
  tr (<) = < **tr (<=) = ≤**
  tr(!=) = ≠ **tr(&) = ∧**
  tr(a or b) = a ∨ b **tr(=>) = →**
  tr(<=>) = ↔
- **Образ условного выражения: tr(c ? a : b) = if(c , a , b)**
- **Образы кванторов следующие**:
  tr(**forall**) = ∀
  tr (**exists**) = ∃
- Образы агрегатов, используемые для задания значений структурных типов, будут рассмотрены вместе со структурными типами.
- Мультипеременные и мультивыражения в системе Russel будем рассматривать как набор соответствующих выражений и переменных
- Элемент массива, модификация массива и структуры будут рассмотрены вместе с описаниями образов соответствующих типов в главе 4.2.

Остается неопределенной только операция XOR (исключающее или), а также операции > и ≥. Определим операцию XOR в библиотеке системы Russell:

```
constant
{
    symbol XOR ;
    ascii XOR ;
}
rule rl_xor (var ph : wff, var ps : wff)
{
```

```
        term : wff = # ( ph XOR ps ) ;
}
definition def_xor (var ph : wff, var ps : wff)
{
        defiendum : wff =  # ( ph XOR ps ) ;
        definiens : wff =  # ( ( ph ∨ ps ) ∧ ¬ ( ph ∧ ps ) ) ;
        ----------------
        prop : wff = |- ( defiendum ↔ definiens ) ;
}
```

Образ операции **xor** определяется следующим образом:

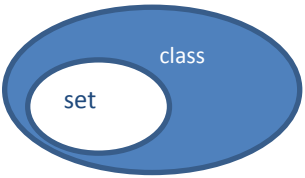
tr(a xor b) = a XOR b //для логических операндов.

Для операций $>$ и $\geq$:

```
constant
{
        symbol > ;
        ascii > ;
        latex > ;
}
rule rl_greater_than ( )
{
        term : class = #  > ;
}
constant
{
        symbol ≥ ;
        ascii ≥ ;
        latex ≥ ;
}
rule rl_ greater_or_equal ( )
{
        term : class = #  ≥ ;
}
definition df_greater_than ( )
{
        defiendum : class =  # > ;
        definiens : class =  # ( ( ℝ* × ℝ* ) \ ≤ ) ;
```

```
        ----------------
        prop : wff = |- ( defiendum = definiens ) ;
}
definition df _ greater_or_equal ( )
{
        defiendum : class =  # ≥ ;
        definiens : class =  # ( ( ℝ* × ℝ* ) \ < ) ;
        ----------------
        prop : wff = |- ( defiendum = definiens ) ;
}
```

4.2. Образы типов



Несмотря на то, что система Russell обладает богатой библиотекой, в ней определены всего три типа:

type wff ; //формула;
type class ; //класс;
type set : class ; //подтип типа класс – тип множество.

Других типов в библиотеке системы Russell нет.

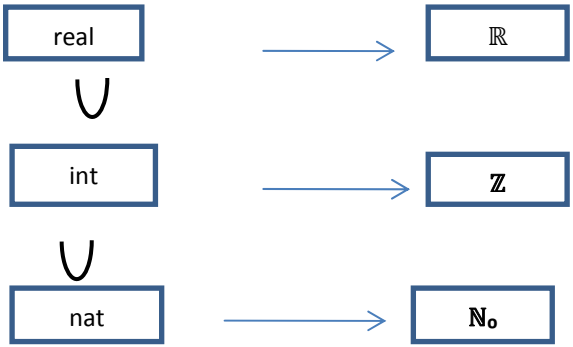
Но язык системы Russell допускает определение типов и их подтипов в исходных текстах библиотек.

4.2.1. Числовые типы

Определим базовые числовые типы языка предикатного программирования в виде соответствующих библиотек системы Russell:

```
type real : set ;    //тип real как подтип типа set
type int : real ;   //тип int как подтип тип real
type nat : int ;    // тип nat как подтип типа int
type bool : nat ;  //тип boll как подтип типа nat
```

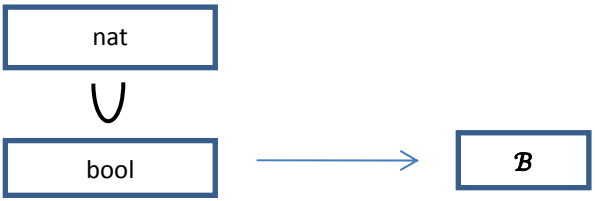
Соответствие между приведенными типами и множествами базисной библиотеки Russell следующее:



Так как множество значений типа **bool** не определено в библиотеке, определим его следующими описаниями:

```
definition df_bool ( var x : set )
{
        defiendum : class =  # 𝔅 ;
        definiens : class =  # { x | ( ( x = 1 ) ∨ ( x = 0 ) ) } ;
        -----------------
        prop : wff = |- ( defiendum = definiens ) ;
}
```

Теперь можем провести аналогичное соответствие:



4.2.2. Массивы

Массивы математически представляются в виде функций, а точнее, сюръекций из множества индексов в множество элементов. Общее определение массива на языке системы Russell:

```
type array : set;
//определили тип массива
constant
{
    symbol Array ;
    ascii Array ;
}
rule rl_Array ( var I : set, var E : set )
{
    term : array = # Array ( I , E ) ;
}
definition def_Array ( var Ar : set, var E : set, var I : set )
{
     defiendum : array =  # Array ( I , E ) ;
     definiens : class =  # { Ar | ( Ar : I ↠ E ) } ;
     ----------------
     prop : wff = |- ( defiendum = definiens ) ;
}
```

Здесь **I** – множество индексов, **E** – множество элементов.

Для большей понятности рассмотрим конкретный пример декларации типа массива на языке предикатного программирования:

```
type 3_to_int = array ( int, 1..3 );
```

Для данного примера после трансляции на языке системы Russell получим:

```
type 3_to_int : set;
constant
{
    symbol 3_to_int_Array ;
    ascii 3_to_int_Array ;
}

rule rl__to_int_Array (var Ar : set)
{
    term : 3_to_int = # 3_to_int_Array ;
}

definition df_3_to_int_Array (var Ar : set)
{
```

```
    defiendum : 3_to_int =  # 3_to_int_Array ;
    definiens : set =  # { Ar | Ar : ( 1 ... 3 ) ↠ ℤ } ;
    -----------------
    prop : wff = |- ( defiendum = definiens ) ;
}
```

Рассмотрим пример инициализации какого нибудь экземпляра данного типа. Для задания значений используется агрегат-массив:

```
3_to_int ArrayTest = [ 3, 1, 4 ];
```

В Russell'е получим:

```
constant
{
    symbol Array_Test ;
    ascii Array_Test ;
}

rule rl_ArrayTest ( )
{
    term : 3_to_int = # Array_Test ;
}

definition df-ArrayTest ( )
{
    defiendum : 3_to_int =  # Array_Test ;
definiens : class =  # { 3_to_int_Array | ( ( 3_to_int_Array ` 2 ) = 1 ) ∧
( (    3_to_int_Array ` 1 ) = 3 ) ∧ ( ( 3_to_int_Array ` 3 ) = 4 ) ) } ;
    -----------------
    prop : wff = |- ( defiendum = definiens ) ;
}
```

Элемент массива представляет собой значение определенной нами функции на соответствующем индексе.

Модификация массива производится присваиванием новых значений соответствующим индексам.

Не будем забывать, что массивы в языке P могут быть многомерными. В таком случае множество индексов будет представляться декартовым произведением $I = I_1 \otimes I_2 \otimes \ldots \otimes I_n$, а инициализация аналогична инициализации одномерных массивов.

4.2.3. Структуры

Тип данных «структура» определяется в виде декартова произведения полей структуры. Структуры имеют произвольное конечное число полей, число которых мы знаем на этапе трансляции. Общее определение структуры с двумя полями будет иметь в Russell'е следующий вид:

```
type struct : set ;

constant
{
symbol Str_Ex ;
ascii Str_Ex ;
}

rule rl_Struct ( var F1 : class , var F2 : class )
{
   term : struct = # Str_Ex ( F1 , F2 ) ;
}
   definition df_Struct ( var F1 : class , var F2 : class )
{
  defiendum : struct = # Str_Ex ( F1 , F2 ) ;
  definiens : class = #   F1 ⊗ F2  ;
  ----------------
  prop : wf f = |- ( defiendum = definiens ) ;
}
```

Здесь **F1,F2 –** это поля структуры.

Как и в случае массивов, рассмотрим конкретный пример декларации типа структуры:

type Point = struct (int x, y);

Тогда в Russell'е будем иметь

```
type Point : set;
constant
{
    symbol Point_Struct;
}
rule rl_Point_Struct ( var x : int, var y : int )
{
    term : Point = # Point_Struct ( x , y ) ;
}
```

```
definition def_Point_Struct ( var x : int, var y : int )
{
    defiendum : Point =  # Point_Struct ( x , y ) ;
    definiens : class =  # { ⟨ x , y ⟩ | ( ( x ∈ ℤ ) ∧ ( y ∈ ℤ ) ) } ;
    ----------------
    prop : wff = |- ( defiendum = definiens ) ;
}
```

Инициализируем экземпляр этого типа конкретными значениями агрегат-стркутуры :

```
Point pt = (1, 2);
```

Получим

```
constant
{
    symbol pt;
}

rule rl_Pt ( )
{
    term : Point = # pt ;
}

definition df_Pt ( )
{
    defiendum : Point =  # pt ;
    definiens : class =  # Point_Struct ( 1 , 2 ) ;
----------------
prop : wff = |- ( defiendum = definiens ) ;
}
```

4.2.4. Множества

Множества языка предикатного программирования – это множество всех подмножеств некоторого конечного базового типа.

```
type t_set : set;

constant
{
    symbol Set;
}
```

```
rule rl_T_Set ( var T : class )
{
    term : t_set = #  Set( T );
}

definition def_T_Set ( var T : class )
{
        defiendum : t_set =  # Set(T) ;
        definiens : class =  # { x | ( x ⊆ T ) } ;
        -----------------
    prop : wff = |- ( defiendum = definiens ) ;
 }
```

Операции же для работы с множествами можем взять из библиотеки.

Образы операций:

tr(+) = (∪)
tr(-) = (\ )
tr(&) = (∩)
tr(in) = (∈)

Рассмотрим пример декларации типа множества (множество целых элементов):

```
type int_set  = set (int);
```

После трансляции будем иметь:

```
type int_set : set;

constant
{
    symbol IntSet ;
    ascii IntSet ;
}

rule rl_IntSet ( )
{
    term : int_set = # IntSet ;
}

definition df_intSet (var x : set)
{
```

```
    defiendum : int_set =  # IntSet ;
    definiens : class =  # { x | ( x ⊆ ℤ ) } ;
    -----------------
    prop : wff = |- ( defiendum = definiens ) ;
}
```

4.2.5. Простые объединения

Объединение (**Union**) является типом, принимающим значение одного из своих конструкторов, каждый из которых состоит из набора полей.

Рассмотрим в качестве примера следующее объединение:

```
type number = union (
    nat_num ( nat n1 ),
    int_num ( int n2 ),
    real_num ( real n3 ),
    complex_num ( real re, real im )
);
number one = nat_num ( 1 );
```

В Russell'е получим набор определений конструкторов и их типов:

```
type number : set;
type nat_num : number;
type int_num : number;
type real_num : number;
type complex_num : number;

constant
{
    symbol Constr_nat_num  ;
    ascii nat_num  ;
}

rule rl_Constr_nat_num (var n1 : nat)
{
    term : nat_num = # Constr_nat_num ( n1 ) ;
}

definition def_Constr_nat_num (var n1 : nat)
{
```

```
    defiendum : nat_num =  # Constr_nat_num ( n1 ) ;
    definiens : class =  # { n1 | ( n1 ∈ ℕ₀ ) } ;
     -----------------
    prop : wff = |- ( defiendum = definiens ) ;
}

/*...определения остальных конструкторов…*/

constant
{
    symbol Constr_complex_num  ;
    ascii Constr_complex_num  ;
}
rule rl_Constr_complex_num (var re : real, var im : real)
{
    term : complex_num = # Constr_complex_num ( re , im ) ;
}
definition def_Constr_complex_num (var re : real, var im : real)
{
    defiendum : complex_num =  # Constr_complex_num ( re , im ) ;
    definiens : class =  # { ⟨ re , im ⟩ | ( ( re ∈ ℝ ) ∧ ( im ∈ ℝ ) ) } ;
-----------------
    prop : wff = |- ( defiendum = definiens ) ;
}
```

Теперь определим множество, соответствующее множеству значений данного типа union:

```
constant
{
    symbol Number_Set  ;
    ascii Number_Set  ;
}

rule rl_Number_Set ( )
{
 term : number = # Number_Set ;
}

definition def_Number_Set (var n1 : nat, ... , var re : real, var im : real)
```

```
{
 defiendum : number =  # Number_Set;
 definiens : class =  # ( Nat_Num ( n1 ) ∪ ( ... ∪ Complex_Num ( re , im ) ) ) );
 ----------------
 prop : wff = |- ( defiendum = definiens );
}
```

Остается оттранслировать декларацию переменной данного типа:

```
constant
{
    symbol number_one ;
    ascii number_one ;
}

rule rl_number_one ( )
{
    term : number= # number_one ;
}

definition def_number_one ( )
{
    defiendum : number =  # number_one ;
    definiens : set =  # Nat_Num ( 1 ) ;
    ----------------
    prop : wff = |- ( defiendum = definiens ) ;
}
```

4.2.6. Списки

Списки будем представлять как упорядоченные пары **⟨h , t⟩**, где h – голова списка (head), имеющая тип значений, хранящимся в списке; t – хвост списка (tail), имеющий тип список. Чтобы эти пары отделить от произвольных упорядоченных пар определим конструктор списка. Заметим, что определяемая нами пара должна иметь тип списка. Для большей наглядности рассмотрим трансляцию на примере целых списков:

```
constant
{
    symbol IntList ;
    ascii IntList ;
```

```
}

rule rl_Int_List ( var h : int, var t : int_list )
{
    term : int_list = # IntList ( h , t ) ;
}

definition def_Int_List ( var h : int, var t : int_list )
{
defiendum : int_list =  # IntList ( h , t ) ;
definiens : class =  # ⟨ h , t ⟩ ;
-----------------
prop : wff = |- ( defiendum = definiens ) ;
}
```

Для описания множества значений данного типа определим множество IntListSet :

```
constant
{
    symbol IntListSet ;
    ascii IntListSet ;
}

constant
{
    symbol Nil ;
    ascii Nil ;
}

rule rl_Nil ( )
{
    term : set = # Nil ;
}

rule rl_Int_List_Set ( var h : int, var t : int_list )
{
    term : int_list = # IntListSet ;
}
```

```
definition def_Int_List_Set ( var h : int, var t : set, var x : set )
{
    defiendum : int_list =  # IntListSet ;
    definiens : class = # ⋂ { x | ( ( Nil ∈ x ) ∧ ∀ t ∈ x ∀ h ∈ ℤ
       ( IntList ( h , t ) ∈ x ) ) } ;
    ----------------
    prop : wff = |- ( defiendum = definiens ) ;
}
```

Заметим, что пустой список описывается константой Nil, которую тоже нужно было определить.

В языке предикатного программированиядля для списков определены функции взятия головы и хвоста. Дадим им определения и в языке Russell.

```
definition def_Get_Head ( var h : int, var t : int_list )
{
defiendum : int =  # GetHead ( IntList ( h , t ) ) ;
definiens : int =  # h ;
----------------
prop : wff = |- ( defiendum = definiens ) ;
}

definition def_Get_Tail (var h : int, var t : int_list)
{
defiendum : int_list =  # GetTail(IntList ( h , t ) ) ;
definiens : int_list =  # t ;
----------------
prop : wff = |- ( defiendum = definiens ) ;
}
```

4.2.7. Прочие типы

Прочие типы являются производными от уже определенных типов.

Тип перечисления является частным случаем объединения с конструкторами без аргументов.

Строки являются частным случаем списка литер.

Необходимо будет также определить образ предикатных типов.

4.2.8. Библиотека образов

Определения различных операций (>, >=, xor) и образов типов (bool, пользовательские типы) были помещены в специальную вспомогательную библиотеку системы Russell. В нее также помещены вспомогательные леммы для нескольких доказанных примеров (см. подробнее в разделе 7).

5. ТРАНСЛЯЦИЯ ТЕОРИЙ

Как уже упоминалось в разделе 3, генерируемые теории на языке внутреннего представления представляются в виде модулей:

module <имя модуля> {
type <имя типа> = …;
formula <имя формулы> (…) = …;
lemma …;
}

Заметим, что описаний типов, формул и лемм может быть несколько.

Напомним, что tr – отображение из языка P в язык Russell.

Образом модуля в Russell'е является следующая конструкция:

tr (module <имя модуля> …) =
theory <имя модуля>;
contents of <имя модуля> {
type <имя типа> = …;
/*описание образа типа (приведено в разделе 5.2.)*/
<образ формулы>;
<образ леммы>
}

Формула на языке внутреннего представления имеет следующий вид:
formula <имя формулы> (<список переменных формулы и их типы>) = <выражение формулы>;

В зыке Russell получим:

constant
{
symbol <имя формулы> **;**
}
rule rl_<имя формулы> **(**<список переменных формулы и их типы>**)**
{
term : wff = # <имя формулы> **(**<список переменных формулы>**) ;**
}

definition def_<имя формулы>
(<список переменных формулы и их типы>**)**
{
defiendum : wff = # <имя формулы>
(<список переменных формулы>**);**
definiens : wff = # <выражение формулы>;
----
prop : wff = | - (defiendum <-> **definiens) ;**
}

5.1. Образ леммы

Лемма на языке внутреннего представления имеет следующий вид:

lemma <утверждение леммы>;

В языке Russell получим декларацию проблемы:

problem Lemma_1
(<список переменных, входящих в утверждение леммы и их типы>**)**
{
prop : wff = |- <утверждение леммы>;
}
proof {
step 1 : wff = ? |- <утверждение леммы>**;**
qed prop 1 = step 1 ;
}

6. ТРАНСЛЯЦИЯ В RUSSELL НА ПРИМЕРАХ

Рассмотрим трансляцию в язык системы Russell на конкретном примере.

Программа умножения через сложение на языке предикатного программирования:

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

Теория после семантического анализа:

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

После трансляции на язык Russel мы получим формулу и проблему (недоказанную лемму):

```
constant
{
    symbol P ;
    ascii P ;
}

rule rl_P ( var a : nat, var b : nat )
{
    term : wff = # P ( a , b ) ;
}
definition def_P ( var a : nat, var b : nat )
{
    defiendum : wff =  # P ( a , b ) ;
    definiens : wff =  # ( ( ( a ∈ ℕ₀ ) ∧ ( b ∈ ℕ₀ ) ) ∧
 ( ( a ≥ 0 ) ∧ ( b ≥ 0 ) ) ) ;
    ----------------
    prop : wff = |- ( defiendum ↔ definiens ) ;
}
problem Ex_1 ( var a : nat,  var b : nat )
{
    hyp : wff = |- ( P ( a , b ) ∧ ( a ≠ 0 ) ) ;
    ----------------
    prop : wff = |- ( ( a - 1 ) ≥ 0 ) ;
}
proof {
    step 1 : wff = ? |- ( ( a - 1 ) ≥ 0 )  ;
    qed prop 1 = step 1 ;
}
```

Далее полученная проблема доказывается системой Russell.

Рассмотрим **второй пример**: программа вычисления наибольшего общего делителя.

```
D(nat a, b:  nat c) ≡ pre a ≥ 1 & b ≥ 1
{          if (a = b) c = a
           else  if (a < b) D(a, b–a: c)
                 else D(a–b, b: c)
} post НОД(c, a, b) measure a + b;
```

После семантического анализа генерируется

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

После трансляции получим

```
definition def_P (var a : nat, var b : nat)
{
 defiendum : wff =  # P ( a , b ) ;
 definiens : wff = # ( ( ( a ∈ ℕ0 ) ∧ ( b ∈ ℕ0 ) ) ∧ ( ( a ≥ 1 ) ∧ ( b ≥ 1 ) ) );
 ----------------
 prop : wff = |- ( defiendum ↔ definiens ) ;
}
problem Lemma_1 (var a : nat, var b : nat)
{
    hyp : wff = |- ( P ( a , b ) ∧ ( a ≤ b ) ) ;
    ----------------
    prop : wff = |- ( ( b - a ) ≥ 0 ) ;
}
proof {
    step 1 : wff = ?|- ( ( b - a ) ≥ 0 )  ;
     qed prop 1 = step 1 ;
}
```

Утверждение второй леммы симметрично утверждению первой, поэтому его рассмотрение можно опустить.

7. АВТОМАТИЧЕСКОЕ ДОКАЗАТЕЛЬСТВО ТЕОРЕМ

Рассмотрим доказательство теорем в системе Russell. Так как система еще находится на стадии доработки, ее возможности ограничены. На дан-

ном этапе он способен доказывать простые утверждения. Рассмотрим пример:

```
problem bijust ( var ph : wff )
{
    prop : wff = |- ¬ ( ( ph → ph ) → ¬ ( ph → ph ) ) ;
}
proof {
step 1 : wff = claim () |- ¬ ( ( ph → ph ) → ¬ ( ph → ph ) ) ;
    qed prop 1 = step 1 ;
}
```

Утверждение автоматически доказывается:

```
theorem bijust ( var ph : wff )
{
    prop : wff = |- ¬ ( ( ph → ph ) → ¬ ( ph → ph ) ) ;
}
proof {
step 1 : wff = claim () |- ¬ ( ( ph → ph ) → ¬ ( ph → ph ) )  ;
proof {
    step 1 : wff = theorem idd () |- ( ( ( ph → ph ) →
¬ ( ph → ph ) ) → ( ph → ph ) )  ;
    step 2 : wff = theorem pm2_01 () |- ( ( ( ph → ph ) →
¬ ( ph → ph ) ) → ¬ ( ph → ph ) )  ;
    step 3 : wff = theorem pm2_65i (step 1, step 2) |-
¬ ( ( ph → ph ) → ¬ ( ph → ph ) )  ;
    qed claim = step 3 ;
    }
    qed prop 1 = step 1 ;
    }
```

Происходит автоматическое доказательство вложенного утверждения (claim). Система сама строит шаги доказательства.

В общем случае доказательство теорем проходит скорее в автоматизированном режиме, а не в автоматическом. Пользователь должен придумать стратегию доказательства, т.е. разбить его на несколько шагов, которые система Russell способна доказать. Пользователю необходимо определять дополнительные леммы, облегчающие процесс доказательства. Помимо этого, в библиотеке необходимо определить леммы, содержащие утверждения о новых определенных для осуществления трансляции объектах.

Для примера рассмотрим доказательство леммы, генерируемой в примере умножения натуральных чисел:

```
theorem Ex_1 ( var a : nat, var b : nat )
{
        hyp : wff = |- ( P ( a , b ) ∧ ( a ≠ 0 ) ) ;
        ----------------
        prop : wff = |- ( ( a - 1 ) ≥ 0 ) ;
}
proof {
  step 1 : wff = theorem pm3_27i (hyp 1) |- ( a ≠ 0 ) ;
  step 2 : wff = definition def_P () |-
    ( P ( a , b ) ↔ ( ( ( a ∈ ℕ0 ) ∧ ( b ∈ ℕ0 ) ) ∧ ( ( a ≥ 0 ) ∧ ( b ≥ 0 ) ) ) ) ;
  step 3 : wff = theorem biimpi (step 2) |-
    ( P ( a , b ) → ( ( ( a ∈ ℕ0 ) ∧ ( b ∈ ℕ0 ) ) ∧ ( ( a ≥ 0 ) ∧ ( b ≥ 0 ) ) ) ) ;
  step 4 : wff = theorem pm3_26i (hyp 1) |- P ( a , b ) ;
  step 5 : wff = axiom ax-mp (step 4, step 3) |-
     ( ( ( a ∈ ℕ0 ) ∧ ( b ∈ ℕ0 ) ) ∧ ( ( a ≥ 0 ) ∧ ( b ≥ 0 ) ) ) ;
  step 6 : wff = theorem pm3_26i (step 5) |- ( ( a ∈ ℕ0 ) ∧ ( b ∈ ℕ0 ) ) ;
  step 7 : wff = theorem pm3_26i (step 6) |- ( a ∈ ℕ0 ) ;
  step 8:wff = theorem pm3_2i (step 7, step 1) |- ( ( a ∈ ℕ0 ) ∧ ( a ≠ 0 ) );
  step 9 : wff = theorem Ex_1_help_3 (step 8) |- ( ( a - 1 ) ≥ 0 ) ;
  qed prop 1 = step 9 ;
}
```

Для данного примера понадобилась следующая вспомогательная лемма:

```
theorem Ex_1_help_1 ( var a : class )
{
    hyp : wff = |- ( ( a ∈ ℕ0 ) ∧ ( a ≠ 0 ) ) ;
}
```

Аналогично в примере вычисления НОД

```
theorem Lemma1 (var a : nat, var b : nat)
{
 hyp : wff = |- ( P ( a , b ) ∧ ( a ≤ b ) ) ;
 ----------------
 prop : wff = |- ( ( b - a ) ≥ 0 ) ;
}
proof {
  step 1 : wff = theorem pm3_26i (hyp 1) |- P ( a , b ) ;
  step 2 : wff = definition def_P () |- ( P ( a , b ) ↔
 ( ( ( a ∈ ℕ0 ) ∧ ( b ∈ ℕ0 ) ) ∧ ( ( a ≥ 1 ) ∧ ( b ≥ 1 ) ) ) ) ;
```

```
  step 3 : wff = theorem biimpi (step 2) |- ( P ( a , b ) →
( ( ( a ∈ ℕ0 ) ∧ ( b ∈ ℕ0 ) ) ∧ ( ( a ≥ 1 ) ∧ ( b ≥ 1 ) ) ) ) ;
  step 4 : wff = theorem pm3_26d (step 3) |- ( P ( a , b ) →
( ( a ∈ ℕ0 ) ∧ ( b ∈ ℕ0 ) ) ) ;
 step 5 : wff = axiom ax-mp(step 1, step 4) |- ( ( a ∈ ℕ0 ) ∧ ( b ∈ ℕ0 ) ) ;
  step 6 : wff = theorem pm3_26i (step 5) |- ( a ∈ ℕ0 ) ;
  step 7 : wff = theorem pm3_27i (step 5) |- ( b ∈ ℕ0 ) ;
  step 8 : wff = theorem nn0rei (step 6) |- ( a ∈ ℝ ) ;
  step 9 : wff = theorem nn0rei (step 7) |- ( b ∈ ℝ ) ;
  step 10 : wff = theorem pm3_27i (hyp 1) |- ( a ≤ b ) ;
  step 11 : wff = theorem subge0i (step 9, step 8) |-
( ( 0 ≤ ( b - a ) ) ↔( a ≤ b ) ) ;
  step 12 : wff = theorem mpbir (step 10, step 11) |- ( 0 ≤ ( b - a ) ) ;
  step 13 : wff = theorem ex2 (step 12) |- ( ( b - a ) ≥ 0 ) ;
  qed prop 1 = step 13 ;
}
```

8. ЗАКЛЮЧЕНИЕ

В работе описаны методы трансляции с языка предикатного программирования на язык Russell, приведены примеры трансляции программ с языка P на язык Russell, а также методы доказательства теорем в системе Russell.

Транслятор формул корректности реализован в составе системы предикатного программирования. Для осуществления трансляции определены образы операций, выражений и типов языка предикатного программирования P в языке системы автоматического доказательства Russell.

На данном этапе образы рекурсивных объединений не разработаны, так как не придумано эффективного способа описания рекурсивных типов данных в системе Russell. Также предстоит построить образы для предикатных типов. Трансляция теорий и их доказательство в системе Russell проводились для нескольких простых тестовых программ на языке P. В дальнейшем необходимо расширить набор тестов для покрытия всех языковых конструкций языка P.

Разработана библиотека системы Russell, содержащая все необходимые для трансляции выражений и типов определения, а также вспомогательные леммы. В процессе доказательства теорем в библиотеку необходимо добав-

лять вспомогательные леммы для упрощения доказательства последующих теорем, что и будет происходить при дальнейшей работе с системой.

СПИСОК ЛИТЕРАТУРЫ

1. Шелехов В. И. Предикатное программирование. Учебное пособие. – НГУ, Новосибирск, 2009. – 109с.
2. Власов Д.Ю. Язык формальной математики Russell // Вестник НГУ. Сер.: Математика, механика, информатика. – 2011. – № 2. – С. 27–50.
3. Wiedijk F. The QED Manifesto Revisited // Studies in logic, grammar and rhethoric. – 2007. – V. 10,No. 23. – P. 121–133. – URL: http://mizar.org/trybulec65/8.pdf
4. Boyer R. et al. The QED Manifesto // Automated Deduction – CADE 12 / Ed. by A. Bundy. – V. 814 of LNAI. – Springer-Verlag, 1994. – P. 238–251. – URL: http://www.cs.ru.nl/~freek/qed/qed.ps.gz
5. The Why platform – URL: http://why.lri.fr/
6. The Coq proof assistant – URL: http://coq.inria.fr/
7. PVS Specification and Verification System – URL: http://pvs.csl.sri.com/
8. The Isabelle proof assistant – URL: http://isabelle.in.tum.de/
9. The Mizar proof assistant – URL: http://mizar.uwb.edu.pl/

В.А. Батраков, В.И. Шелехов

АВТОМАТИЧЕСКОЕ ДОКАЗАТЕЛЬСТВО ФОРМУЛ КОРРЕКТНОСТИ ПРЕДИКАТНОЙ ПРОГРАММЫ В СИСТЕМЕ RUSSELL

Препринт
163

Рукопись поступила в редакцию 10.10.2012
Редактор Т. М. Бульонкова
Рецензент Д.К. Пономарев

Подписано в печать 15.11.2012
Формат бумаги 60 × 84 1/16 Объем 1.8 уч.-изд.л., 2.0 п.л.
Тираж 50 экз.

Центр оперативной печати «Оригинал 2»
г.Бердск, ул. О. Кошевого, 6, оф. 2, тел. (383-41) 2-12-42