УДК 007:681.512.2

ФОРМАЛЬНЫЕ ПРОГРАММНЫЕ МАШИНЫ ДЛЯ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ЯЗЫКОВ: C++

И. Ю. Каширин, д.т.н., профессор кафедры ВПМ РГРТУ Рязань, Россия;
orcid.org/0000-0003-1694-7410, e-mail: igor-kashirin@mail.ru

Рассматривается математический аппарат формальных программных машин, представляющий собой упорядоченное множество универсальных алгоритмических алгебраических систем. Алгебраические системы дают возможность адекватно и однотипно описывать и анализировать программный код для существенно различных языков программирования, таких как C, Lisp, Prolog, Python. Алгебраические системы подразделяются на алгебры работы с оперативной памятью, алгебры констант, алгебры управляющих синтаксических конструкций и алгебры интерпретации программного кода. Сигнатура формальных программных машин позволяет рассматривать композицию и декомпозицию всех элементов программного кода для синтеза простых конструкций в сложные алгоритмические структуры или анализа сложных программ на простейшие конструкции. В статье показывается, что формальные программные машины могут быть использованы не только для необъектных языков, но и для структуризации объектно-ориентированных языков программирования.

Приводится пример объектно-ориентированной программы на языке C++, показывается возможность ее формального анализа и получения объектно-ориентированного кода из функционально-эквивалентных необъектных программ.

В результате произведенных экспериментов показана эффективность применения формальных программных машин для оптимизации программного кода с увеличением эффективности на множестве сложных программ.

Целью работы является презентация эффективного применения формальных программных машин не только для анализа и оптимизации необъектных программ, но и для объектно-ориентированного подхода.

Ключевые слова: *формальные программные машины, алгоритмические алгебры, оптимизация программ, объектно-ориентированное программирование, алгоритмические языки.*

DOI: 10.21667/1995-4565-2024-89-56-64

Введение

Формальные программные машины являются математическим аппаратом, использующим ансамбли алгоритмических алгебраических систем для анализа концептуальной целостности языков программирования [1-3]. Кроме того, они с успехом используются для структуризации (регуляризации) и оптимизации программного кода. Этот математический аппарат позволяет представить принципиально различные языки программирования (например, Lisp, Prolog, C) в единообразной форме, что делает возможным формальное конвертирование конструкций одного языка в другой.

Современные исследования в области алгоритмических алгебр [4-6] показали эффективность формального анализа программ, который в развитых теориях доведен до уровня программных логик [7-10].

До настоящего времени программные машины применялись для программ, не использующих объектно-ориентированный подход. В то же время современные программы в подавляющем большинстве случаев используют концепцию объектов и классов с механизмами инкапсуляции, наследования и полиморфизма. В представленной здесь статье решается актуальная задача проектирования нового метода применения формальных программных ма-

шин для анализа объектно-ориентированных языков и соответствующего программного кода на примере языка программирования C++.

Теоретическая часть

Базовые программные машины

Теория формальных программных машин T_L для языка программирования L в ее наиболее полной форме предполагает наличие шести алгоритмических алгебраических систем (ААС):

$$T_L = < M_L, K_L, A_L, F_L, \Phi_L, I_L >,$$

где M_L – ААС оперативной памяти, K_L – ААС констант языка L, A_L – ААС унарных, бинарных и тернарных операций, F_L – ААС управляющих синтаксических конструкций (операторов), Φ_L и I_L – ААС интерпретации синтаксических конструкций и операций соответственно.

Каждая из алгебраических систем имеет базовое множество-носитель, на котором определяется сигнатура (множество операций), позволяющая строить из более простых элементов сложные производные конструкции:

– базовые и производные типы данных (M_L);

– сложные константы (K_L), соответствующие типам данных;

– сложные последовательности операций (A_L), образующие выражения;

– программы и подпрограммы (F_L), состоящие из операторов;

– перевод всех конструкций языка программирования L в базовую форму интерпретации (Φ_L, I_L), например язык Ассемблера.

Кроме сигнатуры каждая из ААС дополнена множеством отношений, позволяющих упорядочивать конструкции языка L по вложенности друг в друга, величине или размеру, функциональной зависимости, эквивалентности и т.п.

Формальная аналитика и применение программных машин для необъектных языков программирования с целью структуризации и оптимизации подробно рассмотрены в [1-3].

Программные машины для объектно-ориентированных языков программирования

На примере языка C++ рассмотрим концепцию применения формальных программных машин в объектно-ориентированном подходе. Для простоты изложения будем рассматривать сокращенный вариант программной машины T_{C++}, не принимая во внимание ААС интерпретации Φ_{C++} , I_{C++} , поскольку такого формального аппарата будет достаточно для рассмотрения основных языковых конструкций:

$$T_{C++} = < M_{C++}, K_{C++}, A_{C++}, F_{C++} >.$$

Формально можно рассматривать каждый компонент этой системы множеств как алгебру без учета отношений, базовое множество и сигнатура которой зависят от разрабатываемого программистом инструментария и главного модуля программы.

Алгебра оперативной памяти M_{C++} представляется множеством переменных для определенных типов данных, полностью заданных размером занимаемой ими оперативной памяти в байтах.

$$M_{C++} = \{ m_0(size_0), m_1(size_2), \ldots, m_i(size_i), \ldots, m_m(size_m) \},$$

где $size_i \in \{0,1,\ldots, n\}$, то есть принадлежит множеству целых положительных чисел.

Алгебра констант K_{C++} соответствует множеству всех использованных в тексте программы констант разных типов, полностью заданных своим значением.

$$K_{C++} = \{ k_0, k_1, \ldots, k_j, \ldots, k_k\},$$

где j является элементом множества констант C++ , использованных в программе, например, константу 243 можно записать как k_{243}, а литеральную конструкцию «line» обозначить как k_{line}. На взгляд автора, это некоторое упрощение, возможно более строгой записи k_i(«line»), делает формальные выражения алгебры более простыми.

Алгебра A_{C++} состоит из всех унарных, бинарных и тернарных операций языка C++, которые могут быть обозначены, например, так: $a_{=}$, a_{or}, a_{++} и т.п.

Наконец, алгебра F_{C++} базовым множеством имеет все управляющие операторы конкретной программы, например, оператор

$$\text{if}(x == 0)\ y = 1$$

будет в формальной записи выглядеть так:

$$f_{if}(a_{==}(k_0), a_{=}(m_y, k_1)).$$

Безусловно, теоретически существует бесконечное множество всех возможных программ на языке C++. В этом случае каждая из алгебр будет задана бесконечным множеством всех возможных конструкций своего вида, называемых универсумом.

Далее для примера приведем короткую программу на языке C++:

```
class Student
{    char firstName[52];
     char lastName [52];
     int yearBirth;
public:
     Student(const char* firstName, char* lastName, int yearBirth):
            yearBirth(yearBirth) {
          cout << "Student(...) " << lastName << firstName << endl;
          strcpy(this->firstName, firstName);
          strcpy(this->lastName, lastName);
     }
     ~Student(){   cout << "~Student(...) " << lastName << firstName << endl;}
     const char* getLastName() { return lastName; }
     void printAbout(){ cout << getLastName()  << endl; }
};
class Magistrant: public Student{int grades[10];
          public:
     Magistrant(const char* firstName, char* lastName, int yearBirth) :
                              Student(firstName, lastName, yearBirth)  {
          cout << "Magistrant(...) " << lastName << firstName << endl;
          memset(&grades, 0, 10 * sizeof(int));
     }
     ~Magistrant(){cout << "~Magistrant(...) " << getLastName() << endl;}
     void setGrade(int subjectId, int grade)    {
          if(subjectId < 0 || subjectId >= 10 || grade < 0 || grade > 5) return;
          grades[subjectId] = grade;
     }
     int getGrade(int subjectId) {
          if(subjectId < 0 || subjectId >= 10) return 0;
          return grades[subjectId];
     }
     void printGrades() {
          for (int i = 0; i < 10; ++i)      {
               cout << "дисциплина №" << i + 1 << ": ";
               if( grades[i] )
                    cout << " оценка " << grades[i];
               else
                    cout << "нет оценки";
               cout << endl;
          }
     }
};
int main( ){
     Student student("Иван", "Иванов", 2007);
     student.printAbout();   cout << endl;
     Magistrant magistrant("Петр", "Петров", 2004);
     magistrant.setGrade(3, 4);
     magistrant.setGrade(5, 5);
     magistrant.printAbout();
     magistrant.printGrades();
     cout << endl; return 0;
}
```

Эта программа описывает базовый класс студентов «Student» и производный от него класс магистрантов «Magistrant» как частный случай студентов. Объекты (экземпляры) базового класса имеют элементами данных только имя, фамилию и год рождения. Производный класс кроме этих данных содержит еще и оценки обучающегося по десяти дисциплинам.

Если представить эту программу в нотации формальных программных машин, то для главного модуля эти выражения можно представить таблицей 1.

Таблица 1 – Формальное представление главного модуля программы

Table 1 – Formal representation of the main program module

Код программы	Формальное выражение
`int main( ) {`	$f_{def}(f_{main}( ) =$
`Student student("Иван", "Иванов", 2007);`	$f_{Student}(m_{student}, k_{Иван}, k_{Иванов}, k_{2007}) +$
`student.printAbout(); cout << endl;`	$f_{printAbout}(m_{student}) + f_{cout}(k_{endl}) +$
`Magistrant magistrant("Петр", "Петров", 2004);`	$f_{Magistrant}(m_{magistrant}, k_{Петр}, k_{Петров}, k_{2004}) +$

`magistrant.setGrade(3, 4);`
`magistrant.setGrade(5, 5);`
`magistrant.printAbout();`
`magistrant.printGrades();`
	$f_{setGrade}(m_{magistrant}, k_3, k_4) +$
	$f_{setGrade}(m_{magistrant}, k_5, k_5) +$
	$f_{printAbout}(m_{magistrant}) +$
	$f_{printGrades}(m_{magistrant})$

`cout << endl;`	$f_{cout}(k_{endl}) +$
`return 0;}`	$f_{return}(k_0))$

Нетрудно видеть, что такое формальное представление программы не зависит от синтаксических конструкций языка C++. Операторы здесь задаются соответствующими им символами f с именованными индексами, а их последовательность задается операцией «+». Переменные m и константы k также содержат нижние поименованные индексы. Подробнее о формальном представлении программ для других языков программирования можно найти в [2, 3]. Это означает, что запись функционально-эквивалентной программы, например на языке Python или Lisp, в формальной нотации выглядела бы абсолютно тождественно приведенной в таблице 1. По сути дела, этот факт свидетельствует о возможности использовать формальные программные машины для разработки автоматических конверторов программного кода с одного языка на другой.

Фрагмент программы на языке C++, содержащий определения базового и производного классов и использующий кроме уже рассмотренных обозначений конструкции алгебры логики a_{or} и формальной арифметики a_{++}, собран в таблицу 2.

Следует заметить, что в полном соответствии с концепцией функциональности языка программирования C++ в формальных выражениях конструкции создания указателей «*», массивов «[ ]» и оператор присваивания «=» являются операциями соответствующих алгебр формальной программной машины.

Таблица 2 – Формальное представление определений программы

Table 2 – Formal representation of program definitions

Код программы	Формальное выражение

`#include <iostream>`
`#include <strings.h>`
`#include <ctime>`
	$f_{io} ( ) + f_{str} ( ) + f_{ct} ( );$

`class Student`
`{ char firstName[52];`
`char lastName [52];`
`int yearBirth;`
`public:`
	$f_{class} ($
	$m_{Student} = m_{fN}(52) + ml_{N}(52) + m_{yB},$

`Student(const char* firstName, char* lastName, int yearBirth): yearBirth(yearBirth)`	$f_{def}($ $f_{Student}(m_{*fN}(64), m_{*lN}(64), m_{yB}) =$

`cout << "Student(...) " << lastName << firstName << endl;`
`strcpy(this->firstName, firstName);`
`strcpy(this->lastName, lastName);`
	$f_{cout}(k_{Student(...)}+f_{*}(m_{*lN})+ f_{*}(m_{*fN})+m_{endl}))$
	$+f_{strcpy}(m_{this.fN} , m_{fN}) + f_{strcpy}(m_{this.lN} , m_{lN})$
	
	$)$

`~Student(){ cout << "~Student(...) " << lastName << firstName << endl;}`	f_{def} ($f_{\sim student}$( ) = f_{cout}($k_{\sim Student(\dots)}$ + $f_{[]}$(m_{lN})+ $f_{[]}$(m_{fN})+m_{endl}))
`const char* getLastName() { return lastName; }`	f_{def}($f_{getLastName}$( )= f_{reurn}(m_{lN}))
`void printAbout(){ cout<< getLastName() << endl;}`	f_{def}($f_{printAbout}$(f_{cout}($f_{getLastName}$( )+ m_{endl})))
`class Magistrant: public Student`	f_{class}($m_{Student}$ + m_{grades}(10) =
`{cout << "Magistrant(...) " << lastName << firstName << endl; memset(&grades, 0, 10 * sizeof(int)); }`	f_{def}($f_{Magistrant}$ ($f_{Student}$(m_{fN}(64), m_{lN}(64), m_{yB}) + m_{*fN}-(64), m_{*lN}-(64), m_{yB}- + $m_{*grades}$) = f_{cout}($k_{Student(\dots)}$ +f_{*}(m_{*lN})+ f_{*}(m_{*fN}) + m_{endl})) + $f_{mem-set}$($m_{*grades}$,k_0)))
`~Magistrant(){cout << "~Magistrant(...) " << getLastName() << endl;}`	f_{def} ($f_{\sim Magistrant}$( ) = f_{cout}($k_{\sim Magistrant(\dots)}$ + $f_{[]}$(m_{lN})+ $f_{[]}$(m_{fN})+ $f_{getLastName}$( )+m_{endl})))
`void setGrade(int subjectId, int grade)`	f_{def} ($f_{setGrade}$($m_{subjectId}$, m_{grade}) =
`{if(subjectId < 0 || subjectId >= 10 || grade < 0 || grade > 5) return; grades[subjectId] = grade;}`	f_{if} (a_{or}($f_{<}$ ($m_{subjectId}$, k_0), $f_{>=}$($m_{subjectId}$, k_{10}), $f_{<}$ (m_{grade}, k_0), $f_{>}$(m_{grade}, k_5)), $f_{return()}$) + $a_{=}$($m_{*grades+subjected}$, m_{grade})
`int getGrade(int subjectId)`	f_{def} ($f_{getGrade}$($m_{subjectId}$) =
`if(subjectId<0|| subjectId >= 10) return 0; return grades[subjectId];`	f_{if} (a_{or}($f_{<}$($m_{subjectId}$, k_0), $f_{>=}$($m_{subjectId}$,k_{10})), f_{return}($m_{*grades+subjectId}$)))
`void printGrades()`	f_{def}($f_{printGrades}$ () =
`for(int i = 0; i < 10; ++i) { cout << "дисциплина №"<<i + 1 << ": "; if( grades[i] )cout << " оценка " << grades[i]; else cout << "нет оценки"; cout << endl;}`	f_{for}($f_{=}$(m_i, k_0), $f_{<}$ (m_i, k_{10}), a_{++}(m_i), f_{begin}(f_{cout}($k_{дисциплина\ №}$ + k_{i+1} + $k_{:}$) + $f_{if\text{-}else}$($f_{!=}$($m_{*grades+I}$, k_1), f_{cout}($k_{оценка}$ + $k_{*grades+i}$), f_{cout}($k_{нет\ оценки}$, k_{endl}))))

Полезные структурные свойства объектно-ориентированных формальных программных машин

Объектно-ориентированный подход дал возможность разработчикам проектировать программный код более формально и в более жестких рамках синтаксических конструкций (например, C++), чем если бы функционально-эквивалентный код был написан в синтаксисе необъектного языка программирования (например, C). Эту особенность легко заметить в формальных выражениях программной машины. Предположим, есть два фрагмента необъектной программы на языке C для описания абстрактных типов данных Student и Magistrant, представленных следующим описанием:

```
struct Student
{    char FirstNm[52];
     char LastNm [52];
     int YearBirth;
} student;
void Student(const char* FirstNm, char* LastNm, int YearBirth);

struct Magistrant
{    char FirstNm[52];
     char LastNm [52];
     int YearBirth;
            int Grades [10];
} magistrant;
void Magistrant(const char* FirstNm, char* LastNm, int YearBirth);
```

На первый взгляд, описания структур Student и Magistrant в приведенном фрагменте программы имеют существенное отличие, особенно если учесть наименования элементов данных структур. Однако с точки зрения формального подхода названия типов данных не имеют

значения, поскольку в этом случае, как было сказано ранее, учитываются только размеры данных в байтах. Формальная запись фрагментов может выглядеть более унифицировано следующим образом:

f_{def} (m_1 = ($m_{11}(52)$ + $m_{12}(52)$ + $m_{13}(64)$)) + f_{def} ($f_1(m_{11}(52), m_{12}(52), m_{13}(64)$) – для первого фрагмента и

f_{def} (m_1 = ($m_{11}(52)$ + $m_{12}(52)$ + $m_{13}(64)$ + $m_{14}(640)$)) ++ f_{def} ($f_1(m_{11}(52), m_{12}(52), m_{13}(64)$) – для второго фрагмента.

Из этой записи явно следует эквивалентность определений функций f_1 в обоих фрагментах и непустое пересечение (операция «⋂») элементов оперативной памяти данных m_1:

$$m_{11}(52) + m_{12}(52) + m_{13}(64) \bigcap m_{11}(52) + m_{12}(52) + m_{13}(64) + m_{14}(640) =$$
$$= m_{11}(52) + m_{12}(52) + m_{13}(64).$$

При необходимости можно вычислить прямую разность (операция «−») этих элементов памяти:

$$m_{11}(52) + m_{12}(52) + m_{13}(64) + m_{14}(640) - m_{11}(52) + m_{12}(52) + m_{13}(64) = m_{14}(640)$$

Для определения возможности создания оптимизирующей конструкции «базовый класс − подкласс» теперь достаточно следующих двух условий идентификации с помощью формальной записи программного кода.

1. В двух или более фрагментах программы присутствуют элементы памяти, пересекающиеся в их формальной записи.

2. В двух или более фрагментах программы присутствуют однотипные определения функций-подпрограмм. Однотипными с алгебраической точки зрения считаются функции с одинаковым количеством аргументов, полностью совпадающих друг с другом по последовательности типов.

При выполнении указанных условий применимо правило образования наследуемых классов: «выделить общие части элементов данных и функций, определенных на множестве этих элементов данных, в базовый родительский класс; разность элементов данных и функций оформить в виде производного дочернего класса».

Выделение общей части программного кода в единый базовый класс является оптимизирующим преобразованием, сокращающим размер программы. Дополнительным эффектом при определении фрагментов программы в качестве класса является объединение элементов данных совместно с функциями, определенными на множестве этих данных, в единую «капсулу». Инкапсуляция является существенным определяющим фактором эффективности процедуры локализации логических ошибок в программе.

Результирующий программный код при применении сформулированного правила, по сути дела, можно найти в таблице 2.

Условия применения рассмотренных формальных представлений объектно-ориентированных программ полностью соответствуют всем соглашениям подхода к эквивалентной оптимизирующей трансформации необъектного программного кода [2, 3]. Это делает возможным использовать формальную оптимизацию для любых программ на языке C++, включая подмножество базового языка C. К таким оптимизирующим трансформациям относятся:

– исключение повторных вычислений;
– замена сложных фрагментов кода эквивалентным упрощением;
– исключение неработающих переменных;
– замена вычислений константами;
– инициализация переменных;
– сокращение числа обращений к функциям;
– сокращение конструкций, использующих индексы;
– объединение циклов;
– исключение инвариантных вычислений;

– оптимизация условного оператора;
– преобразование размерности данных;
– сокращение логических выражений;
– сокращение числа инициализаций цикла.

Кроме оптимизации программного кода формальные программные машины могут быть использованы в качестве инструментария предварительной токенизации программного кода в языковых нейросетевых моделях [11] для получения языковых конверторов или автоматического поиска логических ошибок.

Экспериментальные исследования

В качестве экспериментальных исследований использовались тексты программ курсовых проектов по дисциплине «Объектно-ориентированное программирование» студентов 3-4 курсов. Были исследованы 42 программы на языке программирования С и 40 программ на языке программирования С++.

Оптимизация студенческих программ производилась с помощью частичной автоматизации для поиска типизированных фрагментов программ, далее в ручном режиме использовались правила эквивалентной трансформации формальных программных машин. Результаты их применения приведены на рисунке 1.

Число строк программного кода (тыс. строк)

Язык С	Оптимизация, %	Язык С++	Оптимизация, %
1	5	1	4
2	5,7	2	4,2
3	6	3	5
4	6	4	6
5	8	5	8
6	9,5	6	9
7	10	7	10,2
8	12	8	12
9	12,9	9	14
10	14	10	17
11	17	11	20
12	21	12	24

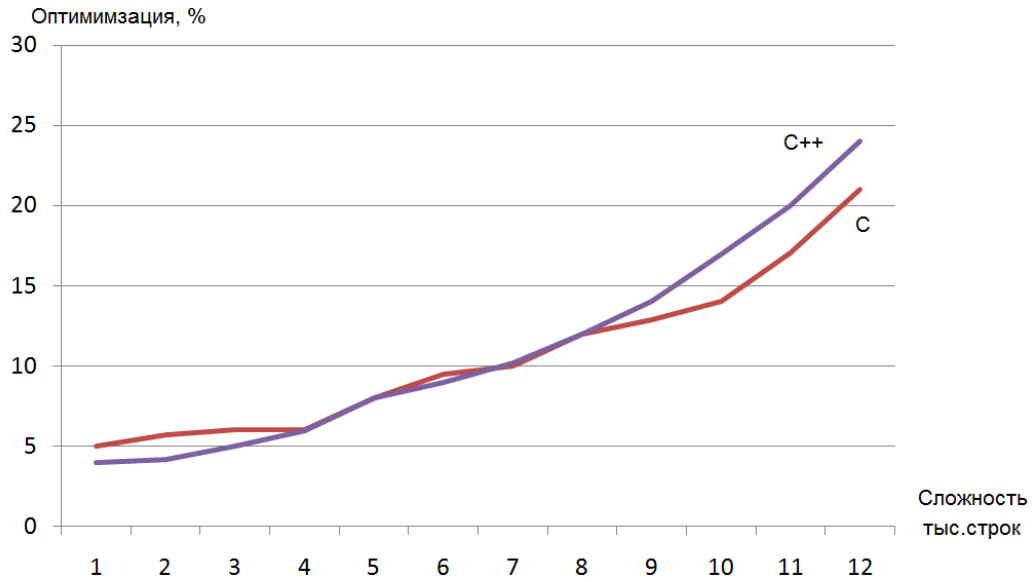

Рисунок 1 – Оптимизация программного кода для языков С и С++
Figure 1 – Optimization of program code for C and C++ languages

Заключение

Произведенные эксперименты позволяют сделать вывод об эффективности использования формальных программных машин не только для программного кода на универсальных, логических и функциональных языках программирования, но и на объектно-ориентированных языках. Формальное представление программ позволяет отыскивать функционально-эквивалентные фрагменты программного кода, находить избыточные и дублирующиеся конструкции, а следовательно, производить их оптимизацию. Для больших программ более 20 тысяч строк такая оптимизация может достигать 20 и более процентов.

Библиографический список

1. **Каширин И.Ю., Лаврентьев С.И., Филатов И.Ю.** Оптимизирующая унификация сложных программ // Вестник Рязанского государственного радиотехнического университета. 2019. № 67. С. 32-44.

2. **Каширин И.Ю., Каширина О.И.** Особенности формального исследования интеллектуального программного инструментария // Вестник Рязанского государственного радиотехнического университета. 2016. № 1. С. 41-54.

3. **Каширин И.Ю., Каширина О.И.** Концептуальная целостность языкового программного инструментария // Вестник Рязанского государственного радиотехнического университета. 2017. № 59. С. 53-65.

4. Algorithmic Algebra. Institute of Algebra and Number Theory. [Электронный ресурс]. URL: https://www.uni-ulm.de/en/mawi/rmath/lehre/wintersemester-2020-21/algorithmic-algebra/ (дата обращения: 05.04.2024).

5. Algorithmic algebra and discrete mathematics. Unikassel Universitet. [Электронный ресурс]. Update date: 12.05.2024 URL: https://www.uni-kassel.de/fb10/en/1000000/institutes/mathematics/research-groups/algorithmic-algebra-and-discrete-mathematics/news-and-events (дата обращения: 25.04.2024).

6. **Bostan A., Strauband A., Yurkevich S.** On the representability of sequences as constant terms // Journal of Number Theory 2023. 253. Pp. 235-256.

7. **Shilov N.V., Yi K.** How to find a coin: propositional program logics made easy // Current Trends in Theoretical Computer Science. 2004. No. 2. Pp. 181-213. (Preliinary version – in The Bulletin of the European Association for Theoretical Computer Science. 2021. No. 75, Pp. 127-151).

8. **Lin Z.** Full nonassociative Lambek calculus with modalities and its applications in type grammars. Adam Mickiewicz University. 2014.

9. **Pshenitsyn T.** Hypergraph Lambek Grammars // Journal of Logical and Algebraic Methods in Programming. 2022. 129. 100798.

10. **Holt L.** Understanding program logic. Victorian Department of Human Service Retrieved. 2009. URL: https://www2.health.vic.gov.au/Api/downloadmedia/%7B4247D1CE-2319-444A-95AB-50483A1757BC%7D.

11. **Каширин И.Ю.** Модели знаний для коррекции дрейфа данных в Data Minig // Вестник Рязанского государственного радиотехнического университета. 2023. № 84. С.108-118.

UDC 007:681.512.2

FORMAL SOFTWARE MACHINES FOR OBJECT-ORIENTED LANGUAGES: C++

I. Yu. Kashirin, Dr. Sc. (Tech.), full professor, RSREU, Ryazan, Russia;
orcid.org/0000-0003-1694-7410, e-mail: igor-kashirin@mail.ru

Mathematical apparatus of formal software machines which is an ordered set of universal algorithmic algebraic systems is considered. Algebraic systems make it possible to adequately and uniformly describe and analyze program code for significantly different programming languages, such as C, Lisp, Prolog, Python. Algebraic systems are divided into algebras of working with RAM, algebras of constants, algebras of control syntactic constructions and algebras of program code interpretation. The signature of formal software machines allows us to consider composition and decomposition of all program code elements for the

synthesis of simple structures into complex algorithmic structures or the analysis of complex programs into simplest structures. The article shows that formal software machines can be used not only for non-object languages, but also for structuring object-oriented programming languages.

An example of an object-oriented program in C++ is given, the possibility of its formal analysis and obtaining object-oriented code from functionally equivalent non-object programs is shown. As a result of the experiments performed, the effectiveness of using formal software machines to optimize program code with increased efficiency on a variety of complex programs is shown.

***The aim** of the work is to present the effective use of formal software machines not only for the analysis and optimization of non-object programs, but also for object-oriented approach.*

***Keywords:** formal software machines, algorithmic algebras, program optimization, object-oriented programming, algorithmic languages.*

DOI: 10.21667/1995-4565-2024-89-56-64

References

1. **Kashirin I.Yu., Lavrent'yev S.I., Filatov I.Yu.** Optimiziruyushchaya unifikatsiya slozhnykh programm. *Vestnik Ryazanskogo gosudarstvennogo radiotekhnicheskogo universiteta*. 2019, no. 67, pp.32-44. (in Russian).

2. **Kashirin I.Yu., Kashirina O.I.** Osobennosti formal'nogo issledovaniya intellektual'nogo programmnogo instrumentariya. *Vestnik Ryazanskogo gosudarstvennogo radiotekhnicheskogo universiteta*. 2016, no. 1, pp. 41-54. (in Russian).

3. **Kashirin I.Yu., Kashirina O.I.** Kontseptual'naya tselostnost' yazykovogo programmnogo instrumentariya. *Vestnik Ryazanskogo gosudarstvennogo radiotekhnicheskogo universiteta*. 2017, no. 59, pp.53-65. (in Russian).

4. Algorithmic Algebra. *Institute of Algebra and Number Theory*. [Electronic resource]. Update date: 02.04.2024 URL: https://www.uni-ulm.de/en/mawi/rmath/lehre/wintersemester-2020-21/algorithmic-algebra/ (date of application: 05.04.2024).

5. Algorithmic algebra and discrete mathematics. *Unikassel Universitet*. [Electronic resource]. URL: https://www.uni-kassel.de/fb10/en/1000000/institutes/mathematics/research-groups/algorithmic-algebra-and-discrete-mathematics/news-and-events (date of application: 25.04.2024).

6. **Bostan A., Strauband A., Yurkevich S.** On the representability of sequences as constant terms. *Journal of Number Theory* 2023, 253, зр. 235-256.

7.**Shilov N.V., Yi K.** How to find a coin: propositional program logics made easy. *Current Trends in Theoretical Computer Science*. 2004, no. 2, pp. 181-213. (Preliinary version – in The Bulletin of the European Association for Theoretical Computer Science. 2021. no. 75, pp. 127-151).

8.**Lin Z.** *Full nonassociative Lambek calculus with modalities and its applications in type grammars*. Adam Mickiewicz University. 2014.

9.**Pshenitsyn T.** Hypergraph Lambek Grammars. *Journal of Logical and Algebraic Methods in Programming*. 2022, 129, 100798.

10. **Holt L.** Understanding program logic. *Victorian Department of Human Service Retrieved*. 2009. URL: https://www2.health.vic.gov.au/Api/downloadmedia/%7B4247D1CE-2319-444A-95AB-50483A1757BC%7D.

11. **Kashirin I.Yu.** Modeli znaniy dlya korrektsii dreyfa dannykh v Data Minig. *Vestnik Ryazanskogo gosudarstvennogo radiotekhnicheskogo universiteta*. 2023, no. 84, pp.108-118.