# ПРИМЕНЕНИЕ СУПЕРКОМПИЛЯТОРА ЯЗЫКА JAVA ДЛЯ РЕШЕНИЯ ОБРАТНЫХ ЗАДАЧ В СТИЛЕ ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ

*Климов Андрей Валентинович*

*(Москва, Институт прикладной математики им. М.В. Келдыша РАН)*

# APPLICATION OF A JAVA SUPERCOMPILER TO INVERSE PROBLEM SOLVING IN THE LOGIC PROGRAMMING STYLE

*Klimov Andrei Valentinovich*

*(Moscow, Keldysh Institute of Applied Mathematics, RAS)*

An approach to solving by a Java supercompiler inverse problems stated in the Java programming language in form of a predicate that checks a solution is demonstrated by the example of the famous N queens puzzle. The supercompiler is a system of equivalence program transformation, program specialization. The solutions to an inverse problem are returned in a transformed (residual) code. The approach was originally suggested by V.F. Turchin for the functional language Refal. It resembles logic programming, but uses a universal object-oriented language rather than a specialized logic programming one. The basic part of the supercompiler, driving, which is used for inverse problem solving, is reviewed.

**Введение.** *Java*-суперкомпилятор *JScp* [5] — это специализатор программ на языке *Java*, построенный на основе метода суперкомпиляции В.Ф. Турчина [3]. При разработке *JScp* метод был расширен [1, 2, 4] на императивные и объектно-ориентированные понятия, отсутствующие в функциональных языках.

В этой статье мы ограничимся обсуждением базовой части суперкомпиляции — *прогонки* [7], — в том виде, как она сейчас реализована для языка *Java* в суперкомпиляторе *JScp*. Прогонка выполняет частичные вычисления над не полностью определенными состояниями *Java*-программы, называемыми *конфигурациями*. Конфигурации аналогичны обычным состояниям *Java*-машины, но кроме конкретных значений содержат *конфигурационные переменные* вместо данных, которые будут известны лишь во время исполнения программы.

В своей классической работе о прогонке [7] В.Ф. Турчин показал, что прогонка программ на языке Рефале дает возможность решать обратную задачу: дана программа-предикат $p$, найти значение $x$, такое что $p(x)$ истинно, — и сформулировал на ее основе «универсальный решающий алгоритм» (УРА). Этот алгоритм аналогичен методу резолюции, встроенному в язык логического программирования Пролог, по удивительному совпадению созданному в том же 1972 году. После появления суперкомпилятора языка Рефал *SCP4* [6] практическая применимость этого подхода была подтверждена экспериментально.

Для решения таких обратных задач принято разрабатывать и использовать функциональные, логические или функционально-логические языки со «списковыми» структурами данных. Однако, он в равной мере применим и к объектно-ориентированным, и другим языкам с другими структурами данных (объектами и т.п.), — лишь бы для них существовал достаточно мощный оптимизатор программ типа суперкомпилятора. Это будет продемонстрировано на примере одной программы на языке *Java* — классической задаче о расстановке ферзей — с использованием нынешней версии суперкомпилятора языка *Java*.

**Терминология.** По традиции, восходящей к А.П. Ершову, программа, выданная суперкомпилятором, называется *остаточной* (*residual*). Остаточная программа функционально эквивалента исходной. Порождение остаточной программы называется *резидуализацией.*

**Решение обратной задачи с помощью оптимизатора программ.** Рассмотрим некоторую функцию-предикат *p*, запрограммированную на языке, для которого имеется мощная система эквивалентных преобразований программ, оптимизатор, — скажем, на языке *Java*:

***static boolean*** *p*(***int*** *x*)

{ <программа возвращающая ***true***, когда *x* удовлетворяет *p*(*x*)> }

Если мы применим *JScp* или другой оптимизатор *Java*-программ, нам может «повезти» и остаточная программа может принять, например, такой вид:

***static boolean*** *p*(***int*** *x*)

{ ***if*** (*x* == 5) ***return true***; ***if*** (*x* == 9) ***return true***; ***return false***; }

По такому коду видно, что предикат *p* истинен при двух значениях аргумента — 5 и 9. В более сложных случаях выходы ***return true*** и ***return false*** могут быть запрятаны в большой код. Чтобы сделать решения более видимыми, вызовем предикат *p* из вспомогательной подпрограммы *pPrint*, печатающей аргумент *x*, если *p*(*x*) истинно:

***static void*** *pPrint*(***int*** *x*)

{ ***if*** (*p*(*x*)) *System.out.println*(*x*); ***return***; }

Просуперкомпилируем метод *pPrint*. Даже если остаточная программа имеет сложный вид, в ней будут хорошо видны вызовы *println* с решениями перед предложениями ***return***:

***static void*** *pPrint*(***int*** *x*)

{ … *System.out.println*(5); ***return***; … *System.out.println*(9); ***return***; … }

Мы можем не вчитываться в код, скрытый многоточиями, а лишь проверить, что предложения *println* в принципе достижимы, и прочесть ответы в аргументах вызова *println*.

Сделаем небольшую техническую модификацию нашего подхода: подпрограмма *p* может не возвращать ***true*** и ***false***, а вырабатывать исключение (скажем, *Error*), если аргумент не удовлетворяет условию задачи, и доходить до выхода ***return***, если удовлетворяет:

***static void*** *pPrint*(***int*** *x*)

{ *p*(*x*); *System.out.println*(*x*); ***return***; }

Отметим, что если не удастся найти в остаточной программе достижимых предложений *println* с решениями, то это не означает, что их нет, а лишь означает, что наша система слаба для решения данной задачи, и можно поставить вопрос о ее совершенствовании.

В некоторых случаях может быть выдан такой остаточный код, что отсутствие решений очевидно, и можно констатировать, что наш оптимизатор доказал это, например:

***static void*** *pPrint*(***int*** *x*)

{ ***throw new*** *java.lang.Error*(); }

// или { ***return false;*** } при оптимизациях исходного предиката *p*

В следующих разделах дается обзор части метода суперкомпиляции, реализованного в *Java*-суперкомпиляторе, — части, достаточной для решения широкого класса обратных задач, а затем демонстрируется его работа на задаче о расстановке ферзей.

**Понятие конфигурации.** Интерпретатор исполняет программу над полностью определенными данными, а суперкомпилятор выполняет программу над частично определенны-

ми данными, над множествами состояний. Процесс выполнения программы над не полностью определенными состояниями в суперкомпиляции называется *прогонкой* [7]. Прогонка очень похожа на интерпретацию с главным отличием, что в тех случаях, когда операцию не удается выполнить, порождается остаточный код (операция *резидуализируется*).

Представление множества состояний суперкомпилируемой программы называется *конфигурацией*. Общий рецепт посторенние понятия конфигурации в суперкомпиляторе из понятия состояния программы в интерпретаторе таково: расширьте область данных *переменными* (называемых *конфигурационными*) и позвольте им встречаться везде, где могут быть обычные данные. Каждая конфигурационная переменная идентифицируется уникальным целочисленным индексом и содержит тип ее значений: либо примитивный тип языка *Java*, либо ссылочный тип (вместе с указанием класса и некоторой дополнительной информацией), либо строчный тип. Конфигурация изображает множество состояний, которые получаются подстановкой всевозможных допустимых значений конфигурационных переменных. Конфигурационные переменные становятся локальными переменными остаточной программы.

В *Java*-машине состояние программы состоит из *глобальных переменных* (***static*** поля классов), представления *нитей* (*threads*) и *кучи* (*heap*). Конфигурации в суперкомпиляторе состоят из тех же элементов, но с особенностями (для обсуждения которых нет места в короткой статье). Дадим определение понятия конфигурации в нынешним *JScp*:

- *конфигурация* — это тройка (*нить*, *ограничения*, *куча*);
- *нить* — это *стек вызовов* — последовательность *фреймов*;
- *ограничения* на конфигурационные переменные в этой статье не используются;
- *фрейм* — это тройка (*локальная среда*, *стек вычислений*, *точка программы*);
- *локальная среда* — это отображение локальных переменных в конфигурационные значения;
- *конфигурационное значение* — это либо обычное значение, либо *конфигурационная переменная*;
- *стек вычислений* выражений — последовательность конфигурационных значений;
- представление *точки программы* несущественно. Достаточно предполагать, что оно позволяет продолжить суперкомпиляцию с данной точки;
- *куча* — отображение ссылочных конфигурационных переменных в представление информации об объекте или массиве, известное в период суперкомпиляции. Оно содержит класс объекта, значения полей, а также информацию, специфичную для суперкомпиляции. Поля и элементы массивов содержат либо конфигурационные значения, либо пометку «значение неизвестно».

*Java*-суперкомпилятор *JScp* выполняет эквивалентное преобразование одного или нескольких методов *Java*-программы. Просуперкомпилированные тела методов замещают исходные. Метод суперкомпилируется с *начальной конфигурации*, состоящей из одного фрейма, в котором формальные параметры связаны с новыми конфигурационными переменными.

**Прогонка вызовов методов.** В нынешней версии *JScp* вызовы методов либо раскрываются (*inlining*), либо *резидуализируются* — переносятся в остаточную программу после подстановки конфигурационных значений в аргументы. Выбор определяется опциями *JScp*, заданными пользователем. Раскрытие метода в суперкомпиляторе выполнятся аналогично

вызову метода в интерпретаторе или раскрытию в обычных компиляторах: к стеку вызовов добавляется новый фрейм, в котором параметры метода связаны со значениями аргументов. По выходу из метода значение выражения в предложении ***return*** (если оно есть) возвращается в вызвавший фрейм как значение вызова метода.

**Прогонка выражений и присваиваний.** Прогонка выражения и оператора присваивания похожа на обычную интерпретацию: текущая конфигурация преобразуется в новую и вырабатывается значение (которое может быть конфигурационной переменной), — а отличается дополнительным порождением остаточного кода, который может быть пустым, если информации хватило, чтобы выполнить все операции как в интерпретаторе.

Для целей данной статьи достаточно считать, что *Java*-выражение состоит из констант, локальных переменных, обращений к полям объектов и элементам массивов, операторов ***new*** создания объекта и массива, унарных и бинарных операций и вызовов методов.

Результат прогонки константы — ее значение. Результат прогонки локальной переменной — конфигурационное значение, связанное с переменной, и пустой остаточный код.

Оператор ***new*** создает новый объект или массив в куче, связывая его с новой ссылочной конфигурационной переменной *v*, и всегда резидуализируется в виде декларации *t v* = ***new*** …, где *t* — тип объекта или массива, поскольку в этом момент неизвестно, придется ли оставить этот объект в остаточной программе или все операции над ним удастся выполнить. Операции доступа к полю объекта или элементу массива и присваивания в них также всегда резидуализируются. Код, оказавшийся лишним, удаляется постпроцессингом.

Унарная и бинарная операция либо вычисляются, если в аргументах достаточно информации, чтобы породить обычное значение, либо резидуализируется с новой конфигурационной переменной *v* в качестве ее значения в виде оператора присваивания *t v* = *e*, где *e* — выражение, представляющее данную операцию с подставленными в нее аргументами.

**Прогонка условных предложений.** Рассмотрим фрагмент исходной программы ***if*** (*c*) *a*; ***else*** *b*; *d*, где *c* — условное выражение, предложения *a* и *b* — ветви условного предложения, а последовательность предложений *d* выполняется по выходу из предложения ***if***.

Если прогонка условного выражения *c* дает значение ***true*** или ***false***, соответствующая ветвь *a* или *b* используется для дальнейшей суперкомпиляции вместо предложения ***if***. В противном случае, прогонка условного выражения *c* порождает остаточный код *c*' (быть может, пустой), конфигурационную переменную $v_c$ в качестве значения условия *c* и две конечные конфигурации $C_t$ и $C_f$, которые станут начальными конфигурациями ветвей *a* или *b* соответственно. В *JScp* в режиме по умолчанию, конфигурации $C_t$ и $C_f$ полагаются равными конфигурации $C_c$, полученной по общему алгоритму прогонки выражения.

Если же включен режим *сужений* (необходимый для решения обратных задач), в тех случаях, когда условное выражение на верхнем уровне имеет операцию сравнения значений примитивных типов на равенство (==) или неравенство (!=), производится *сужение* конфигурации $C_c$ для ветви, соответствующей равным аргументам. А именно, пусть после подстановки аргументов операция сравнения имеет вид *v* == *x* или *v* != *x*, где *v* — конфигурационная переменная, *x* — конфигурационное значение. Тогда конфигурация $C_t$ для равенства (==) и конфигурация $C_f$ для неравенства (!=) формируется подстановкой значения *x* вместо переменной *v* в конфигурацию $C_c$. Условие *c* другого вида сводится к сравнению *c* == ***true***.

По окончании суперкомпиляции двух ветвей *a* и *b* с начальными конфигурациями $C_t$ и $C_f$ выдаются их остаточный код *a*' и *b*' и две конечные конфигурации $C_a$ и $C_b$. Дальнейшая суперкомпиляция предложений *d*, стоящих после условного предложения, может проводиться одним из двух способов, выбираемых пользователем *JScp*: либо путем объединения конфигураций $C_a$ и $C_b$ в одну $C_g$ путем их обобщения и суперкомпиляции *d* с начальной конфигурацией $C_g$, — этот способ для решения обратных задач не используется, поэтому подробнее рассматриваться не будет; либо суперкомпиляцией *d* два раза с начальными конфигурациями $C_a$ и $C_b$, получая остаточный код $d_a$ и $d_b$ соответственно, с выдачей для всего фрагмента остаточного кода вида:

*c*'; ***if*** ($v_c$) { *a*'; $d_a$; } ***else*** { *a*'; $d_b$; }

**Прогонка циклов.** Для решения обратных задач используется самая агрессивная стратегия для циклов: их полная раскрутка, когда это возможно за конечное число раз.

**Пример: задача о расстановке ферзей.** Рассмотрим задачу о нахождении расстановки *N* ферзей на шахматной доске размером *N***N* так, чтобы они не били друг друга.

На рис. 1 изображена исходная программа на языке *Java*, проверяющая, что расстановка ферзей правильная. Доска и операции над ней реализованы в классе *NQueens*. Поле ***int*** *n* — размер стороны доски *N*. Поле ***boolean***[][] *board* — двумерный булевский массив — описывает расстановку ферзей: элемент ***true*** обозначает стоящего в клетке ферзя. Метод *checkNQueens* вместе со вспомогательными методами из левой колонки проверяет, описывает ли массив правильную расстановку ферзей, вырабатывая исключение *Error*, если это не так.

В правой колонке на рис. 1 описаны подпрограммы для реализации описанного выше подхода к нахождению значений элементов массива *board*: вспомогательный метод печати доски *printBoard*, метод *checkAndPrintNQueens*, печатающий аргумент-доску *board*, если он задает правильную расстановку ферзей, и два метода *check4Queens* и *check8Queens*, вызывающие предыдущий метод для досок размером 4*4 и 8*8 соответственно. В методе *main* приведен пример проверки правильной расстановки 4 ферзей на доске 4*4.

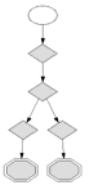
Для поиска расстановки ферзей на досках 4*4 и 8*8 методы *check4Queens* и *check8Queens* суперкомпилируются с самыми агрессивными значениями опций *JScp*, управляющих стратегиями суперкомпиляции, чтобы добиться максимальной развертки дерева остаточной программы (*-invoke* — раскрывать вызовы методов, *-contractEqualPrimary* — включить сужения в операциях сравнения для примитивных типов, *-nojoin* — не объединять ветви по выходу из управляющих конструкций, *-unrollLoopAlways* — раскручивать циклы).

На рис. 2 показан полный текст остаточного кода метода *check4Queens*, в том виде в каком он выдан суперкомпилятором *JScp* (руками лишь изменена разбивка на строки и вставлены некоторые пробелы). Код состоит из дерева проверок элементов массива *board*, на двух листьях которого печатаются два решения — две расстановки 4 ферзей на доске 4*4.

Код остаточной программы метода *check8Queens* с решением задачи 8 ферзей на доске 8*8 привести в статье невозможно: его длина почти 4 тыс. строк. Он был выдан суперкомпилятором *JScp* за 5 мин. на *Pentium* 4 1.8 Ггц. На рис. 2 изображен его граф, также выданный суперкомпилятором. Это блок схема остаточного кода, в которой удаленны ветви, на которых вырабатываются исключения предложениями вида ***throw new*** *java.lang.Error*(). Этот граф является деревом с 92 листьями со всеми 92 решениями задачи о 8 ферзях.

```
public class NQueens {
  final int n; // размер N стороны доски
  final boolean[][] board; // доска N*N
// создание объекта-доски из массива
  NQueens(int n, boolean[][] board) {
    this.n = n;
    this.board = new boolean[n][n];
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        this.board[i][j] = board[i][j];
  }
// проверка ферзей на доске N*N
  void checkNQueens() {
    for (int i = 0; i < n; i++) {
      int j = findQueenInRow(i);
      noQueensInRowToTheRight(i, j);
      noQueensInColumnBelow(i, j);
      noQueensInLeftDiagonal(i, j);
      noQueensInRightDiagonal(i, j);
  } }
// поиск в строке i индекса клетки с ферзем
  int findQueenInRow(int i) {
    for (int j = 0; j < n; j++)
      if(board[i][j]) return j;
    throw new Error();
  }
// проверки, что ферзя нет справа, ниже,
// по диагоналям влево вниз и вправо вниз
// относительно текущей позиции (i, j)
  void noQueensInRowToTheRight(int i, int j) {
    for (int k = j+1; k < n; k++)
      if (board[i][k]) throw new Error();
  }
  void noQueensInColumnBelow(int i, int j) {
    for (int k = i+1; k < n; k++)
      if (board[k][j]) throw new Error();
  }
  void noQueensInLeftDiagonal(int i, int j) {
    for (int k = 1;
         i+k < n && 0 <= j-k && j-k < n;
         k++)
      if (board[i+k][j-k]) throw new Error();
  }
  void noQueensInRightDiagonal(int i, int j) {
    for (int k = 1; i+k < n && j+k < n; k++)
      if (board[i+k][j+k]) throw new Error();
  }

// печать доски
  void printBoard() {
    for (int i = 0; i < n; i++) {
      String s = "";
      for (int j = 0; j < n; j++)
        s += board[i][j] ? "*" : ".";
      System.out.println(s);
    }
  }

// проверка и печать доски N*N
  static void checkAndPrintNQueens(
    int n, boolean[][] board)
  {
    System.out.println("board "+n+"*"+n);
    NQueens b = new NQueens(n, board);
    b.checkNQueens();
    b.printBoard();
  }

// проверка доски 4*4
  static void check4Queens(boolean[][] b) {
    checkAndPrintNQueens(4, b);
  }

// проверка доски 8*8
  static void check8Queens(boolean[][] b) {
    checkAndPrintNQueens(8, b);
  }

// головная программа:
// пример вызова проверки доски
// с правильной расстановкой 4 ферзей
  public static void main(String[] args) {
    check4Queens(
      new boolean[][]{
        new boolean[]{false, true , false, false},
        new boolean[]{false, false, false, true },
        new boolean[]{true , false, false, false},
        new boolean[]{false, false, true , false},
      });
  }
}
```

Рис. 1. Исходная программа *c* классом *NQueens*, реализующим доску *N*N* в виде двумерного булевского массива *board*, в котором значения ***true*** обозначают стоящих в данных клетках ферзей. Метод *checkNQueens* проверяет, что массив описывает правильную расстановку ферзей, вырабатывая исключение *Error*, если это не так.



```
static void check4Queens
                    (final boolean[][] b_1) {
 java.lang.System.out.println("board 4*4");
  final boolean b_0_0_13 = b_1[0][0];
  final boolean b_0_1_17 = b_1[0][1];
  final boolean b_0_2_21 = b_1[0][2];
  final boolean b_0_3_25 = b_1[0][3];
  final boolean b_1_0_37 = b_1[1][0];
  final boolean b_1_1_41 = b_1[1][1];
  final boolean b_1_2_45 = b_1[1][2];
  final boolean b_1_3_49 = b_1[1][3];
  final boolean b_2_0_61 = b_1[2][0];
  final boolean b_2_1_65 = b_1[2][1];
  final boolean b_2_2_69 = b_1[2][2];
  final boolean b_2_3_73 = b_1[2][3];
  final boolean b_3_0_85 = b_1[3][0];
  final boolean b_3_1_89 = b_1[3][1];
  final boolean b_3_2_93 = b_1[3][2];
  final boolean b_3_3_97 = b_1[3][3];
  if (b_0_0_13) {
    throw new java.lang.Error();}
  if (b_0_1_17) {
    if ( b_0_2_21 || b_0_3_25 || b_1_1_41
      || b_2_1_65 || b_3_1_89 || b_1_0_37
      || b_1_2_45 || b_2_3_73 || !b_1_3_49
      || b_3_3_97 || b_2_2_69 || !b_2_0_61
      || b_3_0_85 || !b_3_2_93) {
      throw new java.lang.Error();}
    java.lang.System.out.println(". * . .");
    java.lang.System.out.println(". . . *");
    java.lang.System.out.println("* . . .");
    java.lang.System.out.println(". . * .");
    return;}
  else {
    if (!b_0_2_21 || b_0_3_25 || b_1_2_45
      || b_2_2_69 || b_3_2_93 || b_1_1_41
      || b_2_0_61 || b_1_3_49 || !b_1_0_37
      || b_3_0_85 || b_2_1_65 || !b_2_3_73
      || b_3_3_97 || !b_3_1_89) {
      throw new java.lang.Error();}
    java.lang.System.out.println(". . * .");
    java.lang.System.out.println("* . . .");
    java.lang.System.out.println(". . . *");
    java.lang.System.out.println(". * . .");
    return;}}
```

Рис. 2. Остаточный код метода *check4Queens*, в котором в аргументах *println* звездочками закодированы два решения задачи о ферзях для доски 4*4. Справа вверху показана блок-схема этого кода без ветвей с предложениями ***throw*** — остаточный граф, выданный *JScp*.

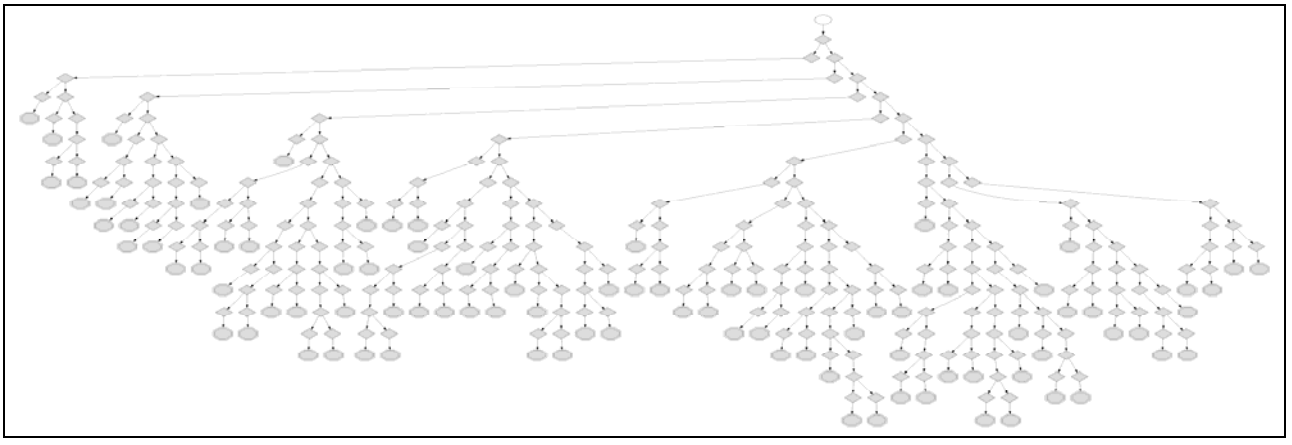


Рис. 3. Остаточный граф (блок-схема) метода *check8Queens* с 92 листьями, в которых представлены 92 решения задачи о расстановке 8 ферзей на доске 8*8.

**Заключение.** Был продемонстрирован метод решения обратных задач, описанных на языке *Java*, с помощью суперкомпилятора *JScp*, так, что исходная программа задает не перебор и поиск решений, а лишь проверку одного решения. Выдачу всех решений в виде кода остаточной программы организует суперкомпилятор, который является универсальным инструментом эквивалентного преобразования программ в отличие от интерпретаторов языков логического программирования, в которых «зашит» частный метод поиска решений. Конечно, класс обратных задач, которые способен решить суперкомпилятор, зависит от мощности реализованных в нем методов. Прогонка в нынешнем суперкомпиляторе *SCP4* языка Рефал [6] мощнее унификации в языке *Prolog*. Прогонка в *JScp* пока слабее, но тем не менее уже пригодна для решения некоторого класса задач.

Такой подход к решению обратных задач с использованием суперкомпилятора не является новым. Он был предложен В.Ф. Турчиным [7] и реализован А.П. Немытых для функционального языка Рефал [6]. Вклад нашей работы — практическая демонстрация применимости этого подхода к объектно-ориентированному языку *Java*, который не несет в себе ничего от языков логического программирования, с помощью разработанного нами суперкомпилятора *JScp*.

Работа поддержана грантами РФФИ № 08-07-00280-а и № 09-01-00834-а.

ЛИТЕРАТУРА

1. Klimov And.V. An approach to supercompilation for object-oriented Languages: the Java Supercompiler case study // *Proceedings of the First International Workshop on Metacomputation in Russia* (July 2–5, 2008, Pereslavl-Zalessky, Russia). — Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008. — P.43–53. — http://meta2008.pereslavl.ru/accepted-papers/paper-info-4.html.
2. Klimov And.V. A Java supercompiler and its application to verification of cache-coherence protocols // Amir Pnueli, Irina Virbitskaite, Andrei Voronkov (eds.), *Procedings of the 7th International Andrei Ershov Memorial Conference "Perspectives of Systems Informatics", PSI 2009, Akademgorodok, Novosibirsk, Russia, June 5–19, 2009*. Novosibirsk: Ershov Institute of Informatics Systems, 2009.
3. Turchin V.F. The concept of a supercompiler // *ACM Transactions on Programming Languages and Systems* 8, N 3, 1986. — P.292–325.
4. Климов Анд.В. Особенности построения суперкомпилятора языка Java // *Научный сервис в сети Интернет: решение больших задач: Труды Всероссийской научной конференции* (21–27 сентября 2008 г., г. Новороссийск). — М.: Изд-во Московского университета, 2008. — С.252–256.
5. Климов Анд.В., Климов Арк.В., Шворин А.Б. *Проект Java-суперкомпилятор*. Электронный ресурс. — http://www.supercompilers.ru.
6. Немытых А.П. *Суперкомпилятор SCP4: общая структура*. — М.: Editorial URSS, 2007.
7. Турчин В.Ф. Эквивалентные преобразования рекурсивных функций, описанных на языке Рефал // *Труды симпозиума "Теория и методы построения систем программирования"*, Киев-Алушта). 1972. — С.31–42.