Специализация интерпретаторов на объектно-ориентированных языках может быть эффективной

Игорь А. Адамович

Институт программных систем

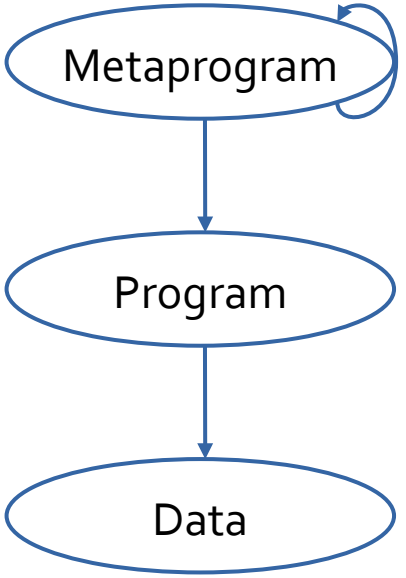
им. А.К.Айламазяна РАН

8 октября 2022

Что такое метавычисления?

«Мы должны научиться манипулировать компьютерными программами так, как мы сейчас обращаемся с числами на Фортране».

В. Ф. Турчин, 1971



self-application

Примеры метавычислений

Примеры манипуляции программами:

- **Компиляторы, интерпретаторы** и другие обработчики языков того же типа.
- **Различные анализаторы программ** (напр. используемые в компиляторах)

Метавычислениями тот или иной метод называется тогда, когда он достаточно «нетривиален», глубок и сложен.

Специализация программ
- Частичные вычисления
- Суперкомпиляция
- Частичная дедукция

Сплавление программ
- Дефорестация
- Суперкомпиляция
- Частичная дедукция

Инверсия программ
- Суперкомпиляция
- Частичная дедукция

Верификация
- Различные методы

Понятие специализации программ

Специализация — это оптимизация программ на основе использования наперёд заданной информации о значении части переменных.

Пусть дана программа f(x,y) от двух аргументов x и y и значение одного из ее аргументов x=a.

Результатом специализации программы f(x,y) по известному аргументу x=a называется новая программа от одного аргумента g(y), обладающая следующим свойством: f(a,y)=g(y) для любого y.

Специализация – это одна из задач метавычислений.

Частичные вычисления

- ***Частичные вычисления*** – это один из методов специализации программ.
- Специализация методом частичных вычисления разделяет программные конструкции на ***статические*** (зависящие от известных данных) и ***динамические*** (зависящие от неизвестных данных).
- Операции над статическими данными выполняются, а над динамическим переносятся в результирующую (***остаточную***) программу.
- Остаточная программа зависит только от неизвестных (на стадии специализации) данных.

Этапы частичных вычислений

Первый этап частичных вычислений называется ***анализ времени связывания*** (Binding Time Analysis, BTA). Этот этап отвечает за разделение операций и данных на статические и динамические.

Вторая стадия ответственна за исполнение статической части программы и перенос динамической части в отдельную программу. Этот этап называется ***генератор остаточной программы*** (Residual Program Generator, RPG). На этом этапе, по существу, и происходят частичные вычисления.

Терминологическое замечание

В теории частичных вычислений части исходного кода, которые будут выполнены при специализации называются ***статическими***.

Остальные части исходного кода, которые переносятся в остаточную программу, называются ***динамическими***.

Термин ***статический*** конфликтует с модификатором статический (`static`) в Java, а термин ***динамический*** может быть перепутан с аналогичным термином «времени исполнения». Поэтому далее мы будем избегать использования этих слов по отношению к частичным вычислениям и будем использовать символы `S` и `D`, например `S`-аннотация, `D`-аннотация, `S`-код, `D`-код, `S`-часть и `D`-часть программы.

Моновариантный и поливариантный BT-анализ

BT-анализ может обладать свойством моновариантности или поливариантности. ***Моновариантный*** анализ отличается тем, что каждый фрагмент кода получает однозначную разметку S или D. ***Поливариантный*** BT-анализ позволяет одному и тому же участку кода иметь несколько разметок.

BT-анализ может быть:

- ***Поливариантным по переменным*** — разрешается размечать различные вхождения одной переменной по-разному.
- ***Поливариантным по операциям*** — одному участку кода исходной программы могут сопоставляться несколько участков выходной размеченной программы с разными BT-разметками.
- ***Поливариантным по методам (подпрограммам)*** — каждый метод может быть размечен в зависимости от его использования, в зависимости от разметки аргументов и результатов.
- ***Поливариантным по классам*** — каждый класс может иметь несколько разметок.

Пример: специализация функции Аккермана

Математическое определение:

$$A(m,\ n) = \begin{cases} n+1, & m = 0; \\ A(m-1,\ 1), & m > 0,\ n = 0; \\ A(m-1,\ A(m,\ n-1)), & m > 0,\ n > 0. \end{cases}$$

Программа на языке Java:

```java
package examples;

import ru.psiras.spec.annotations.Specialize;
import ru.psiras.spec.annotations.SpecKeep;

public class AckermannExample {

    @SpecKeep
    public static final long A (long x, long y) {
        if (x == 0) {
            return y + 1;
        } else if (y == 0) {
            return A(x - 1, 1);
        } else {
            return A(x - 1, A(x, y - 1));
        }
    }

    @Specialize
    public static final long test(long y) {
        return A(3, y);
    }
}
```

Пример: результат специализации

Результат специализации:

```
public class AckermannExample {
    public long test(long y) {
        return A_3(y);
    }

    public final long A_3(long y) {
        if (y == 0) return A_2(1);
        else return A_2(A_3(y - 1));
    }

    public final long A_2(long y) {
        if (y == 0) return A_1(1);
        else return A_1(A_2(y - 1));
    }

    public final long A_1(long y) {
        if (y == 0) return A_0(1);
        else return A_0(A_1(y - 1));
    }

    public final long A_0(long y) {
        return y + 1;
    }
}
```

Программа на языке Java:

```
1  package examples;
2
3  import ru.psiras.spec.annotations.Specialize;
4  import ru.psiras.spec.annotations.SpecKeep;
5
6  public class AckermannExample {
7
8      @SpecKeep
9      public static final long A (long x, long y) {
10         if (x == 0) {
11             return y + 1;
12         } else if (y == 0) {
13             return A(x - 1, 1);
14         } else {
15             return A(x - 1, A(x, y - 1));
16         }
17     }
18
19     @Specialize
20     public static final long test(long y) {
21         return A(3, y);
22     }
23 }
```

Специализация арифметики

```
1  package examples.spec;
2
3  import ru.psiras.spec.annotations.Specialize;
5
6  public class example0 {
7      @Specialize
8      void method(int intD0, int intD1) {
9          int a, b, c;
10         a = 5; b = 6;
11         c = a + b;
12         c = a + intD0;
13         c = intD0 + intD1;
14     }
15 }
```

Специализация if с D-условием

```
1  package examples;
2
3  import ru.psiras.spec.annotations.Specialize;
4
5  public class example0 {
6
7      @Specialize
8      void method (boolean D) {
9          int a, b;
10         if (D) {
11             a = 5;
12         } else {
13             a = 6;
14         }
15         b = a;
16     }
17 }
```

Объединение веток if с S-условием

```
1  package examples.spec;
2
3
4  import ru.psiras.spec.annotations.Specialize;
6
7  public class example0 {
8      @Specialize
9      void method(int intD) {
10         int a, b;
11         boolean S = true;
12         if (S) {
13             a = 5;
14         } else {
15             a = intD;
16         }
17         b = a;
18     }
19 }
```

Первая проекция Футамуры-Турчина

По определению специализации: $f(a, y) = g(y)$ (1)

В качестве f выберем интерпретатор: $I_S(p_L, d)$

В качестве S-аргумента (a) - программу: P_L

В качестве g(y) — остаточную программу: Q_S

Тогда $SPEC(I_S, P_L)(d) \stackrel{def}{=} Q_S(d) =_{(1)} I_S(P_L, d) \equiv P_L(d)$

для любого d.

По сути, Q_S – это сильно оптимизированный интерпретатор I_s, который умеет исполнять только одну программу – P_L.

Первая проекция Футамуры (формально)

- Дан специализатор **SPEC(f_S(x,y),a)** для программ **f_S(x,y)** на языке **S**.
- Дан интерпретатор **I_S(p_L(x),d)**, написанный на этом языке **S**, для программ **p_L(x)** на языке **L.**
- Дана некоторая программа **P_L(x)** на языке **L**.

Если применить специализатор **SPEC** к интерпретатору **I_S** языка **L**, при этом в качестве известного аргумента интерпретатора передать программу **P_L** на языке **L**, получится новая программа **Q_S**: **$Q_S \stackrel{def}{=}$ SPEC(I_S,P_L)**

Тогда:

$$Q_S \stackrel{def}{=} SPEC(I_S,P_L) \equiv P_L$$

Вторая проекция Футамуры-Турчина

По определению специализации: $f(a, y) = g(y)$ (1)

В качестве f выберем специлизатор: SPEC(f, s)

S-аргумент (a) - интерпретатор: $I_S(p_L, d)$

В качестве g(y) — остаточную программу: C

Тогда $SPEC(SPEC, I_S)(P_L) \stackrel{def}{=} C(P_L) =_{(1)} SPEC(I_S, P_L) = Q_S$
(1-я проекция)

Если на вход C подать программу P_L, то получим ее откомпилированную версию.

По сути, C – это сильно оптимизированный специализатор SPEC, который умеет оптимизировать только один интерпретатор – I_S.

Третья проекция Футамуры-Турчина

По определению специализации: $f(a, y) = g(y)$ (1)

В качестве f выберем специлизатор: SPEC(f, s)

S-аргумент (a) - интерпретатор: SPEC(f, s)

В качестве g(y) — остаточную программу: CoGen

Тогда $SPEC(SPEC, SPEC)(I_S) \stackrel{def}{=} CoGen\,(I_S) =_{(1)} SPEC(SPEC, I_S) = C_S$ (2-я проекция)

Если на вход CoGen подать интерпретатор I_S, то получим компилятор.

По сути, CoGen – это сильно оптимизированный специализатор SPEC, который умеет оптимизировать сам себя.

Содержание исследования

Настоящая работа сфокусирована на вопросе эффективного применения специализации (и метода частичных вычислений в особенности) для ***компиляции программ без компилятора***, но с использованием интерпретаторов, ***написанных на объектно-ориентированных языках***.

Идеи компиляции программы без компилятора называются проекциями Футамуры–Турчина

Интерпретируемая программа

В качестве интерпретируемой программы $\mathbf{P_L}$ выбрана программа вычисления квадратного корня с заданной точностью ***методом Ньютона***. Математическая формула, на которой основывается вычисление такова:

$$x_{i+1}=\frac{1}{2}\left(x_i+\frac{a}{x_i}\right)$$

Псевдокод программы (переменные ε (точность) и a (подкоренное число) являются параметрами данной программы):

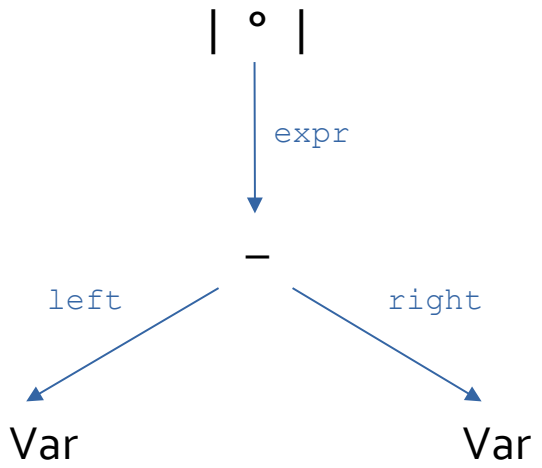
```
x := 1
d := a
x := 0.5 * (x + a/x)
while (ε < |x - d|) begin
    d := x
    x := 0.5 * (x + a/x)
end
```

Интерпретаторы

Рассматриваются два интерпретатора, построенные на классических принципах: на основе **рекурсивного спуска** и на основе **шаблона «посетитель»**, свойственного объектно-ориентированному программированию.

Программа в интерпретаторы и специализатор поступает в виде ***дерева абстрактного синтаксиса*** (AST).

Пример дерева абстрактного синтаксиса для выражения |x - d|



Интерпретатор, работающий рекурсивным спуском

Рекурсивный спуск имеет императивные корни и обычно не использует возможностей, характерных для объектно-ориентированных языков.

В интерпретаторе, работающем рекурсивным спуском, описаны ***взаимно-рекурсивные функции*** для каждой конструкции языка. Например, для того чтобы вычислить значение интерпретируемого предложения, вызывается функция обработки предложения processStatement.

processStatement определяет конкретный тип предложения и, основываясь на этом конкретном типе, вызывает соответствующую функцию обработки конкретной конструкции.

```java
public class ASTInterpreter {

    // ... some code above

    public static final double processStatement(Statement st,
            Var vars) {
        double result = 0;
        if (st.type == ASSIGNMENT) {
            return processAssignment((Assignment)st, vars);
        } else if (st.type == WHILE){
            return processWhile((While)st, vars);
        }
        return result;
    }

    // ... some code below
```

Интерпретируемая программа:

```
x := 1
d := a
x := 0.5 * (x + a/x)
while (ε < |x - d|) begin
     d := x
     x := 0.5 * (x + a/x)
end
```

Интерпретатор, работающий рекурсивным спуском (2)

```java
public class ASTInterpreter {

    //... some code above

    public static final double
    processArithExpression(ArithExpression expression,
                           Var vars) {
        if (expression.type == VALUE) {
            return value((Numeric)expression);
        } else if (expression.type == OPERATOR) {
            return operator((ArithOperator)expression, vars);
        } else if (expression.type == VARIABLE) {
            return variable((Variable)expression, vars);
        }  else if (expression.type == ABS) {
            return abs((Abs)expression, vars);
        }
        return 0;
    }

    private static double processAssignment(Assignment assignment, Var vars) {
        Var varIter = lookup(vars, assignment.left);
        varIter.value = processArithExpression(assignment.right,
                                               vars);
        return varIter.value;
    }

    //...some code bellow
```

```java
public class ASTInterpreter {

    // ... some code above

    public static final double processStatement(Statement st,
            Var vars) {
        double result = 0;
        if (st.type == ASSIGNMENT) {
            return processAssignment((Assignment)st, vars);
        } else if (st.type == WHILE){
            return processWhile((While)st, vars);
        }
        return result;
    }

    // ... some code below
```

Интерпретируемая программа:

```
x := 1
d := a
x := 0.5 * (x + a/x)
while (ε < |x - d|) begin
    d := x
    x := 0.5 * (x + a/x)
end
```

Интерпретатор на основе шаблона «посетитель»

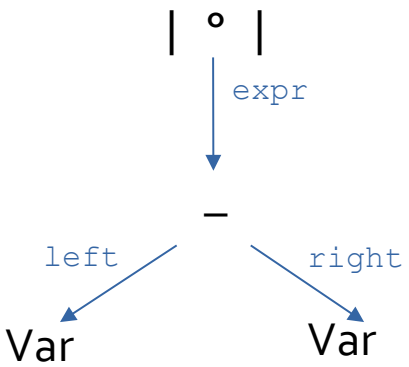
«Посетитель» во многом основывается на полиморфизме – свойстве, присущем объектно-ориентированным языкам.

ASTVisitor содержит объявление метода visit(Type) для каждого возможного класса Type узла в абстрактном синтаксическом дереве.

Каждый класс Type, описывающей тот или иной тип узла в AST, реализует свой виртуальный метод accept(ASTVisitor visitor), который возвращает управление объекту visitor с помощью виртуального вызова visitor.visit(this).

```
1  public abstract class ASTVisitor {
2      public abstract void visit(Assignment assignment);
3      public abstract void visit(Begin begin);
8      public abstract void visit(While wh);
9
10     public abstract void visit(Abs abs);
11     public abstract void visit(Numeric value);
12     public abstract void visit(Variable variable);
13     public abstract void visit(Div div);
14     public abstract void visit(Minus minus);
15     public abstract void visit(Mult mult);
16     public abstract void visit(Sum sum);
17
18     public abstract void visit(BooleanValue boolValue);
19     public abstract void visit(Equals equals);
20     public abstract void visit(Greater greater);
21     public abstract void visit(GreaterEquals greaterEquals);
22     public abstract void visit(Less less);
23     public abstract void visit(LessEquals lessEquals);
24 }
```

Интерпретатор на основе шаблона «посетитель» (пример)




```java
public class Abs extends ArithExpression {

    //... some code above

    public void accept(ASTVisitor visitor) {
        if (visitor == null) {
            throw new IllegalArgumentException();
        }
        visitor.visit(this);
    }

    // some code bellow
```

```java
public class ASTInterpreter extends ASTVisitor {

    // ... some code above

    @Override
    public void visit(Abs abs) {
        abs.expr.accept(this);
        currentValue = Math.abs(currentValue);
    }

    // ... some code below
```

```java
public abstract class ASTVisitor {
    public abstract void visit(Assignment assignment);
    public abstract void visit(Begin begin);
    public abstract void visit(While wh);

    public abstract void visit(Abs abs);
    public abstract void visit(Numeric value);
    public abstract void visit(Variable variable);
    public abstract void visit(Div div);
    public abstract void visit(Minus minus);
    public abstract void visit(Mult mult);
    public abstract void visit(Sum sum);

    public abstract void visit(BooleanValue boolValue);
    public abstract void visit(Equals equals);
    public abstract void visit(Greater greater);
    public abstract void visit(GreaterEquals greaterEquals);
    public abstract void visit(Less less);
    public abstract void visit(LessEquals lessEquals);
}
```

Пример специализации выражения |x - d|

Исходные варианты:

Рекурсивный спуск:

```
1 private static double abs(Abs abs, Var vars) {
2     double result = processArithExpression(abs.expr, vars);
3     return Math.abs(result);
4 }
```

Шаблон «посетитель»:

```
1  public class ASTInterpreter extends ASTVisitor {
2
3      // ... some code above
8
9      @Override
10     public void visit(Abs abs) {
11         abs.expr.accept(this);
12         currentValue = Math.abs(currentValue);
13     }
14
15     // ... some code below
```

Остаточная программа:

```
1  accept28: {
2      visit28: {
3          accept30: {
4              visit30: {
5                  accept32: {
6                      visit32: {
7                          obj40_currentValue = obj0_value;
8                      }
9                  }
10                 double left0 = obj40_currentValue;
11                 accept34: {
12                     visit34: {
13                         obj40_currentValue = obj2_value;
14                     }
15                 }
16                 double right = obj40_currentValue;
17                 obj40_currentValue = left0 - right;
18             }
19         }
20         obj40_currentValue = Math.abs(obj40_currentValue);
21     }
22 }
```

Пример специализации цикла while

```
1  public static double whileSpec(While wh, double a, double x, double eps, double d)
2      double result = 0;
3      double sub = x - d;
4      double abs = Math.abs(sub);
5      double leftEps = eps;
6      boolean iterExpression = leftEps < abs;
7      while(iterExpression) {
8          d = x;
9          double sumX = x;
10         double upperA = a;
11         double bottomX = x;
12         double div = upperA / bottomX;    // a / x
13         double sum = sumX + div;          // x + a/x
14         double k = 0.5;
15         x = k * sum;                      //0.5 * (x + a/x)
16         result = x;
17
18         sub = x - d;
19         abs = Math.abs(sub);
20         leftEps = eps;
21         iterExpression = leftEps < abs;
22     }
23     return result;
24 }
```

Интерпретируемая программа:

```
. . .

while (ε < |x - d|) begin
    d := x
    x := 0.5 * (x + a/x)
end
```

Измерение производительности

В экспериментах мы измеряли среднее время выполнения исходных и специализированных версий интерпретаторов в зависимости от точности, с которой необходимо извлечь квадратный корень.

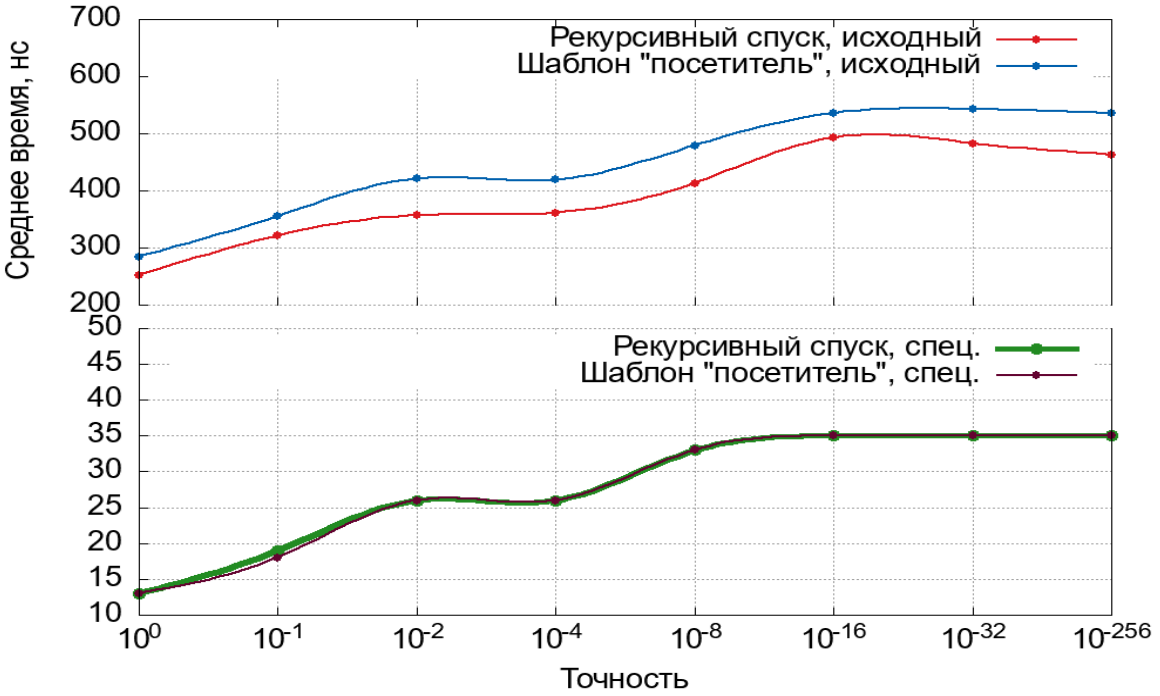
Использовался инструмент Java Microbenchmark Harness (JMH). JMH позволяет выполнять нано/микро/мили/макро тесты производительности для Java программ, избегая при этом различных проблем измерений, которые создает Java-машина.

1. Производился разогрев Java-машины, в течение которой 15 секунд циклически вызывался интерпретатор с заданными параметрами, при этом время выполнения не засекалось. Эта фаза необходима, чтобы Java-машина выполнила все свои внутренние оптимизации программы, что позволяет избежать большой погрешности измерений.

2. В течение 50 секунд опять циклически вызывался интерпретатор. Для каждого отдельного вызова интерпретатора измерялось время его выполнения.

3. Шаги 1 и 2 повторялись 3 раза. После чего вычислялось среднее время выполнения интерпретатора, основываясь на данных с шага 2.

Результаты измерений



Ускорение:
рекурсивного спуск – 12 до 19 раз;
посетитель – 14 до 22 раза.

Относительная погрешность измерений для каждого из замеров не превосходила 3.3%

Заключение

В докладе рассмотрены:

- Постановку задачи специализации.
- Пример компиляции без компилятора, основанный на первой проекции Футамуры-Турчина.
- Способ и результаты измерения производительности исходной и остаточной программы.

- Высокое ускорение получено за счет успешного разрешения полиморфизма, свойственного объектно-ориентированным языкам, на этапе статического анализа.

Заключение

В докладе рассмотрены:

- Постановку задачи специализации.
- Пример компиляции без компилятора, основанный на первой проекции Футамуры-Турчина.
- Способ и результаты измерения производительности исходной и остаточной программы.

- Высокое ускорение получено за счет успешного разрешения полиморфизма, свойственного объектно-ориентированным языкам, на этапе статического анализа.

Спасибо за внимание!