И.А. Адамович, Ю.А. Климов

Специализация интерпретаторов на объектно-ориентированных языках может быть эффективной

Рекомендуемая форма библиографической ссылки

Адамович И.А., Климов Ю.А. Специализация интерпретаторов на объектно-ориентированных языках может быть эффективной // Научный сервис в сети Интернет: труды XXIV Всероссийской научной конференции (19-22 сентября 2022 г., онлайн). — М.: ИПМ им. М.В.Келдыша, 2022. — С. 3-24.
https://doi.org/10.20948/abrau-2022-18
https://keldysh.ru/abrau/2022/theses/18.pdf

Специализация интерпретаторов на объектно-ориентированных языках может быть эффективной

И.А. Адамович[1], **Ю.А. Климов**[2]

[1] *Институт программных систем им. А.К. Айламазяна РАН*
[2] *Институт прикладной математики им. М.В. Келдыша РАН*

Аннотация. Барьеры на пути специализации реальных программ, написанных в объектно-ориентированной парадигме, часто могут быть преодолены при помощи современных методов метавычислений. Один из барьеров — необходимость разрешения полиморфизма на этапе анализа программы, до ее исполнения. Эта проблема успешно решается для ряда случаев в специализаторе JaSpe, что показано в данной статье. Работа посвящена компиляции программ с использованием метода специализации, без использования компилятора. Мы применили специализатор JaSpe, основанный на методе частичных вычислений, к двум интерпретаторам языка арифметических выражений, написанным на Java. Интерпретаторы были реализованы методом рекурсивного спуска и с использованием шаблона «посетитель». В результате успешной специализации данных интерпретаторов по программе вычисления квадратного корня на языке арифметических выражений были получены скомпилированные версии программы на языке Java. При этом скорость полученных версий программы по сравнению с исходной увеличилась в 12-22 раза.

Ключевые слова: интерпретаторы, компиляторы, частичные вычисления, специализация, метавычисления.

Specialization of interpreters written in object-oriented languages can be effective

I.A. Adamovich[1], **Yu.A. Klimov**[2]

[1] *Program Systems Institute of RAS*
[2] *Keldysh Institute of Applied Mathematics of RAS*

Abstract. Barriers of real object-oriented program specialization can be often overcome using modern metacomputation techniques. One of the barriers is the resolution of polymorphism at the stage of program analysis before the execution of the program. The last problem is successfully solved for a number of cases in the JaSpe specializer, as shown in this paper. The paper is devoted to the program compilation by specialization methods, without the use of a compiler. We have applied the partial evaluator JaSpe to two arithmetic expression language interpreters written in Java. The interpreters were implemented using the recursive descent method and the visitor pattern. As a result of the successful specialization of these interpreters by the square root program written on arithmetic expression language, compiled versions of the latter were obtained. In this case, the acceleration was from 12 to 22 times.

Keywords: interpreters, compilers, partial evaluation, specialization, metacomputations.

1. Вступление

Задача, рассматриваемая в настоящей статье, относится к области специализации программ. Пусть дана некоторая программа **P**$_L$ на некотором языке **L**, часть аргументов которой известна на этапе оптимизации, а другая часть аргументов неизвестна. Используя известную часть аргументов, требуется построить более эффективную программу **Q**$_L$ на том же языке **L**, которая зависит только от неизвестной части аргументов. В данном контексте под эффективностью понимается то, что **Q**$_L$ исполняется быстрее и/или потребляет меньше памяти.

Отметим, что программы **P**$_L$ и **Q**$_L$ – это программы на одном языке **L,** и они должны быть эквивалентны с учетом ограничения, предполагающего, что часть аргументов **P**$_L$ известна.

Специализация программы может быть осуществлена различными методами, например частичными вычислениями [1, 2], суперкомпиляцией [3–5], методом повышения арности функций [6] и другими.

Настоящая статья сфокусирована на вопросе эффективного применения метода специализации для компиляции программ без компилятора, но с использованием интерпретаторов, написанных на объектно-ориентированных языках. Идеи компиляции программы без компилятора называются проекциями Футамуры–Турчина и были впервые предложены Есихико Футамурой [7, 8] и независимо открыты позже Валентин Федоровичем Турчиным [9].

Поскольку в большинстве случаев интерпретатор разработать проще чем компилятор, специализация позволяет получить более легкий способ компиляции программ. Специализация, по мнению авторов данной статьи, имеет перспективы в разработке новых и/или сложных языков программирования.

Цель нашей работы показать, что компиляция с помощью первой проекции Футамуры может быть достаточно эффективной даже в случае объектно-ориентированного подхода к написанию интерпретаторов.

В своей работе мы основываемся на экспериментальном специализаторе JaSpe [10–12]. Этот инструмент работает методом частичных вычислений и погружен в популярную среду разработки Eclipse IDE [13]. Eclipse IDE позволяет нашему специализатору привычным для программиста способом в интерактивном режиме составлять задание на специализацию, получать промежуточные и конечный результаты работы. JaSpe специализирует программы, написанные на широко распространенном объектно-ориентированном языке Java, возвращая в качестве результата программы на том же языке Java.

В отличие от других работ в этой области, специализатор JaSpe применяется до момента исполнения специализируемой программы (Ahead-of-Time, AOT); работает с объектно-ориентированным программами; а также он является инструментом общего назначения, т. е. он способен специализировать не только интерпретаторы, но и любые другие программы.

Важным для настоящей работы свойством нашего специализатора является способность для ряда случаев выполнять на этапе анализа динамическую диспетчеризацию виртуальных вызовов. Это преимущество JaSpe получено в результате адаптации и развитии методов, разработанных в специализаторе CILPE [14, 15], и отличает данные частичные вычислители от предшественников. Существенным отличие JaSpe от CILPE является то, что JaSpe осуществляет преобразование программ из языка Java в язык Java, а CILPE из промежуточного языка CIL платформы .NET в тот же язык CIL. Использование байт-кода не позволяет программисту легко проанализировать результат специализации. В отличие от CILPE, специализатор JaSpe ориентирован на построение результата в человеко-читаемом виде на языке Java.

В настоящей работе мы концентрируемся на примерах применения специализации и не рассматриваем конкретные использованные алгоритмы, методы и принципы частичных вычислений – их можно найти в работах [11, 12].

Структура оставшейся части статьи следующая. В разделе 2 производится обзор области. В разделе 3 подробно объясняется первая проекция Футамуры, теоретическая основа статьи. Раздел 4 описывает то, какую программу мы выбрали в качестве аргумента интерпретаторам. Раздел 5 посвящен интерпретаторам, задействованным в исследовании, их исходному и специализированному коду. В разделе 6 приводятся экспериментальные данные, полученные в результате измерения скорости работы различных версий интерпретаторов. Раздел 7 завершает данную статью и содержит основные выводы.

2. Обзор области

По нашим данным, первое построение компилятора из интерпретатора было получено Моррисом в 1970 году [16] ручным способом.

Первым из известных нам специализаторов, реализовавшим автоматическое порождение компиляторов, является MIX [17]. MIX специализировал программы на функциональном языке Mixwell. Работа [17] стимулировала дальнейшие исследования в области генерации компиляторов.

MIX был первым специализатором, который позволил реализовать на практике проекции Футамуры, в частности сгенерировать компилятор компиляторов. Однако, этот компилятор компиляторов получился громадных размеров и было непонятно, как он реально работает. В специализаторе Unmix [18, 19] были предложены и реализованы два усовершенствования: (1) использование разных представлений для S- и D-значений при генерации остаточной программы и (2) автоматический повышатель арности/местности [20]. В результате, размер генератора компиляторов уменьшился на порядок, а его структура стала очевидной.

Консель и Данви получили автоматическим способом компилятор с исходного языка похожего на Алгол в язык Scheme [21]. Для автоматической генерации использовался частичный вычислитель Schism. Такой подход показал ускорение около сотни раз, если сравнивать интерпретацию программ на исходном языке с их скомпилированными версиями.

Позднее, Консель и Ху автоматически сгенерировали компилятор языка Prolog в язык Scheme на основе специализатора Schism [22]. Скомпилированные таким компилятором программы работали примерно в 6 раз быстрее чем интерпретируемые.

Современный специализатор, использующийся для компиляции без компилятора, встроен в фреймворк Truffle, компонент GraalVM [23, 24]. Truffle предназначен для реализации высокопроизводительных динамических или предметно-ориентированных языков. Специализатор в Truffle предоставляет разработчикам интерпретаторов легкий способ реализации JIT-компиляции для их языков. При этом сам специализатор предназначен только для JIT-компиляции интерпретаторов и не может быть применен к программам другого типа или для компиляции перед исполнением программы.

Алгоритм CompGen [25] сокращает время JIT-компиляции в GraalVM. Идея, которая лежит в основе CompGen – специализировать специализатор интерпретаторов, лежащий в основе Truffle, по конкретному интерпретатору. Однако, Compgen применяется только к некоторой части специализатора, ответственной за оптимизацию определенных узлов в интерпретаторе. Выбор этих узлов осуществляется на основе данных профилирования.

Использованный в нашей работе специализатор JaSpe развивает и адаптирует для языка Java методы реализованные в частичном вычислителе CILPE [14, 15]. CILPE предназначен для специализации программ на языке SOOL: подмножестве языке CIL — промежуточного языка платформы Microsoft .NET.

3. Первая проекция Футамуры

Рассмотрим одну из базовых идей, лежащих в основе компиляции без компилятора. Эти идеи названы проекциями Футамуры-Турчина [7–9], мы остановимся только на первой из них.

Идея первой проекции Футамуры заключается в следующем. Пусть дан специализатор **SPEC($\mathbf{f_S}$(x,y),a)** для программ **$\mathbf{f_S}$(x,y)** на языке **S** (здесь индексом будем обозначать язык, на котором написана данная программа). Предполагаем, что специализируемая программа **$\mathbf{f_S}$(x,y)** (первый аргумент **SPEC**) имеет два аргумента **x** и **y**, она специализируется при известном значении своего первого аргумента **x = a** (второй аргумент **SPEC**), и результатом специализации является некоторая программа также на языке **S**. Также пусть дан интерпретатор **$\mathbf{I_S}$($\mathbf{p_L}$(x),d)**, написанный на этом языке **S**, для программ **$\mathbf{p_L}$(x)** на языке **L**. Также, пусть дана некоторая программа **$\mathbf{P_L}$(x)** на языке **L**. Тогда, если применить специализатор **SPEC** к интерпретатору **$\mathbf{I_S}$** языка **L**, при этом в качестве известного аргумента интерпретатора передать программу **$\mathbf{P_L}$** на языке **L**, получится новая программа **$\mathbf{Q_S}$**: **$\mathbf{Q_S} \stackrel{\text{def}}{=}$ SPEC($\mathbf{I_S}$,$\mathbf{P_L}$)**. Эта программа **$\mathbf{Q_S}$** функционально эквивалентна программе **$\mathbf{P_L}$.** А именно, для любых исходных данных **D**, либо обе программы не завершаются, либо обе программы завершаются и **$\mathbf{Q_S}$(D)** = **$\mathbf{P_L}$(D)**, т.е. эти программы выдают одинаковый результат. Важно, что **$\mathbf{Q_S}$** является программой, написанной на языке **S**, за счет этого ее выполнение более эффективно, чем если бы исходная программа **$\mathbf{P_L}$** исполнялась универсальным интерпретатором **$\mathbf{I_S}$**.

Итак, вместо программы **$\mathbf{P_L}$**, написанной на языке **L**, получена программа **$\mathbf{Q_S}$** на языке **S**, которая выполняет ту же функцию, что и **$\mathbf{P_L}$**. Т. е. мы фактически *скомпилировали программу* **$\mathbf{P_L}$** из языка **L** в язык **S**.

Таким образом, первую проекцию Футамуры можно записать как:

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

4. Интерпретируемая программа

В нашей работе в качестве интерпретируемой программы выбрана программа вычисления квадратного корня с заданной точностью методом Ньютона [26, 27]. Математическая формула, на которой основывается вычисление такова:

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

Псевдокод программы представлен на листинге 1.

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

Листинг 1. Вычисление квадратного корня методом Ньютона.

Переменные ε (точность) и a (подкорневое число) являются параметрами данной программы.

Считается, что предложение while имеет значение последнего предложения в его теле, т. е. для нашего случая while имеет значения равное значению переменной x.

Аналогично while, результат работы программы равен значению последнего обработанного предложения. В итоге результат работы программы на листинге 1 равен значению while.

В исследуемые специализатор и интерпретаторы данная программа поступает в виде дерева абстрактного синтаксиса (AST).

5. Исходный и остаточный код интерпретаторов

В настоящей статье рассматриваются два интерпретатора, построенные на классических принципах: на основе рекурсивного спуска и на основе шаблона «посетитель», свойственного объектно-ориентированному программированию. Метод рекурсивного спуска и шаблон «посетитель» в этой статье не описываются. Мы сделаем акцент на иерархии вызовов функций (для краткого сопоставления подходов) и на динамической диспетчеризации, которая является основной трудностью для применения методов специализации. Описание рекурсивного спуска на примере синтаксического анализа можно найти в [28], а описание шаблона посетитель в [29].

Сделаем терминологическое замечание: в дальнейшем изложении будем называть Java-методы имеющие квалификатор static в их определении (т. е. принадлежащие классу, а не объекту), *функциями*.

Ссылка на git с интерпретаторами, рассматриваемыми в данной работе, размешена в конце статьи.

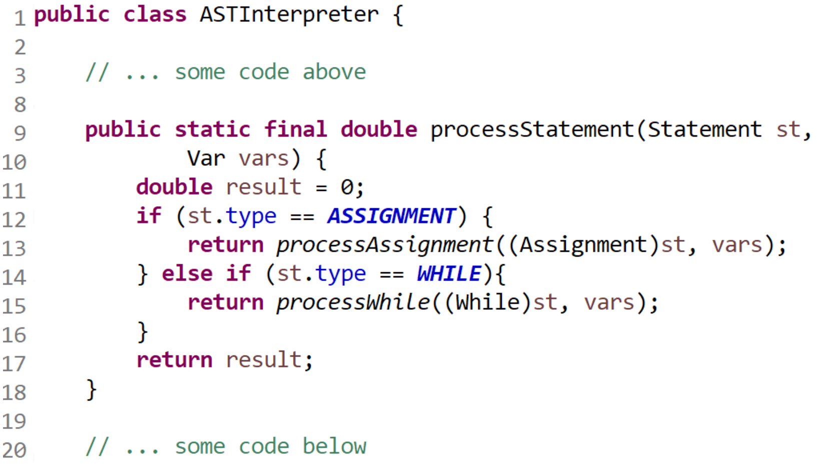
5.1. Рекурсивный спуск

Способ построения интерпретаторов на основе рекурсивного спуска имеет императивные корни и обычно не использует возможностей, характерных для объектно-ориентированных языков. В нашем случае интерпретатор, построенный на основе рекурсивного спуска, описан классом ASTInterpreter, в котором описаны взаимно-рекурсивные функции для каждой конструкции языка. Например, для того чтобы вычислить значение интерпретируемого предложения, вызывается функция обработки предложения:

double ASTInterpreter::processStatement(Statement st, Var vars).

Функция processStatement проста (см. рис. 1) – она определяет конкретный тип предложения и, основываясь на этом конкретном типе, вызывают соответствующую функцию обработки конкретной конструкции. Например для цикла «while» функцией обработки конкретной конструкции является

double ASTInterpreter::processWhile(While wh, Var vars).



```
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
```

Рис. 1. Функция processStatement интерпретатора, работающего рекурсивным спуском

Определение конкретного типа конструкции можно реализовать двумя способами:

1 На основе поля type, помещенного в объект-аргумент функции. Это поле заполняется при создании данного объекта (вызове конструктора).

2 С помощью конструкции obj instanceof type, которая в языке Java проверяет, имеет ли объект obj тип type. Выполнив данную проверку для каждого из типов конструкций, можно определить тип данного объекта-аргумента

Мы выбрали первый способ, потому что он традиционно использовался в императивных реализациях.

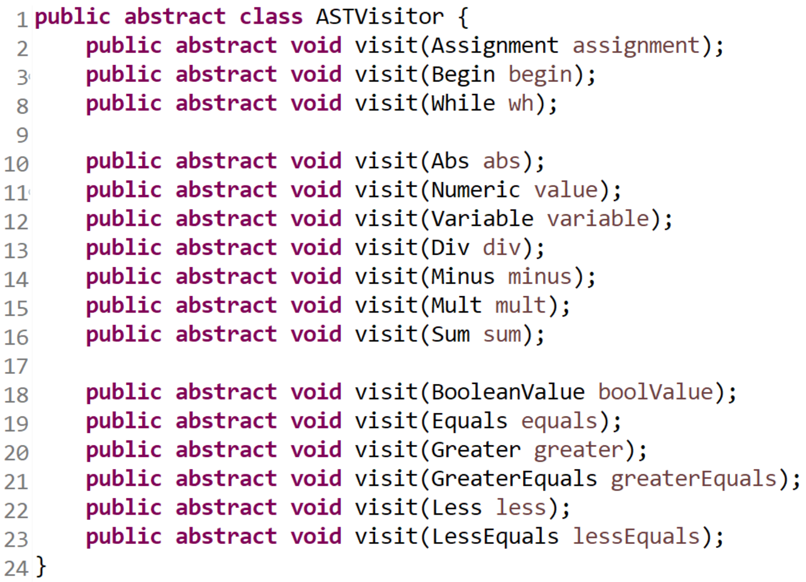
5.2. Шаблон «посетитель»

Интерпретатор, построенный на основе шаблона «посетитель» во многом основывается на полиморфизме – свойстве, присущем объектно-ориентированным языкам.

Интерпретатор в шаблоне «посетитель» – это объект класса ASTInterpreter, унаследованного от ASTVisitor. ASTVisitor содержит объявление метода visit(Type) для каждого возможного класса Type узла в абстрактном синтаксическом дереве (Abstract Syntax Tree, AST) (Type наследует ASTNode). ASTVisitor является абстрактным классом, который содержит только объявления методов, а их реализация перенесена в класс ASTInterpreter. Код класса ASTVisitor, содержащий прототипы всех методов для посещения всех возможных типов узлов AST, помещен на рис. 2.

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).

Пример метода accept для класса While изображен на рис. 3 (accept в остальных классах, описывающих узлы AST, текстуально повторяет метод с рис. 3). Примеры реализации методов visitor.visit(…) можно найти ниже по ходу статьи (рис. 6 и рис. 9).

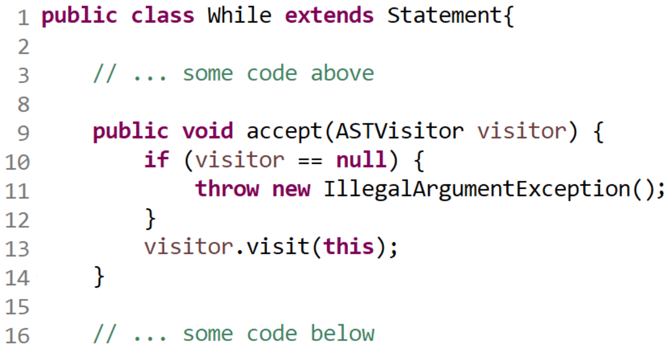


```java
public abstract class ASTVisitor {
    public abstract void visit(Assignment assignment);
    public abstract void visit(Begin begin);
    public abstract void visit(While wh);

    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);
}
```

Рис. 2. Объявление методов в классе ASTVisitor.



```java
public class While extends Statement{

    // ... some code above

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

    // ... some code below
```

Рис. 3. Метод accept в классе While

Когда в некотором методе visit(...) необходимо обработать узел AST subNode, visitor производит вызов метода subNode.accept(this). Последний вызов задействует встроенную в объектно-ориентированные языки процедуру, которая называется динамическая диспетчеризация (dynamic dispatch).

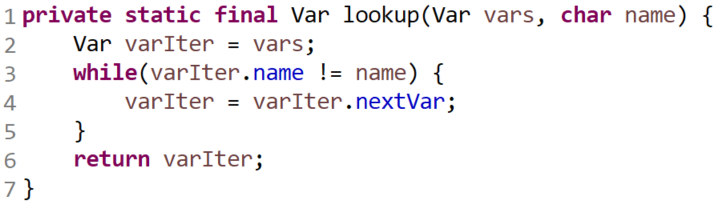
Например, если переменная subNode имеет тип Statement, но в данной переменной хранилась ссылка на объект класса While (такое возможно в Java, если класс While унаследован от типа Statement), то будет вызван метод accept(ASTVisitor visitor), определенный именно в классе While. Метод accept, расположенный в классе While, в соответствии с идеей шаблона «посетитель», вызовет метод visitor.visit(While), что приведет процедуру интерпретации в метод visit(While) интерпретатора. Таким образом мы перешли из некоторого метода visit(...), к анализу подконструкции While в методе visit(While).

Обычно динамическая диспетчеризация производится во время выполнения программы. Однако, разработанный в рамках нашей работы [11, 12, 14, 15] подход позволяет зачастую осуществлять динамическую диспетчеризацию в процессе статического анализа, т. е. до выполнения программы. Такие оптимизации могут существенно улучшать эффективность специализации для объектно-ориентированных языков. Выполнение динамической диспетчеризации до реального исполнения программы роднит специализатор JaSpe с частичным вычислителем CILPE и отличает от предшественников. Однако, JaSpe превосходит CILPE в следующих возможностях: анализ в JaSpe охватывает более широкий набор конструкций, в том числе конструкции-циклы, а также применим к объектам расположенным не на стеке, а в куче.

Исследуемые в данной статье интерпретаторы достаточно объемны, чтобы рассматривать весь их исходный код. Остановимся на нескольких показательных фрагментах интерпретаторов: поиска значения переменной, которая встретилась в арифметическом выражении; анализе арифметических выражений; анализе цикла While.

5.3. Специализация функция поиска значения переменной

Рассмотрим функцию поиска значения переменной и специализированную версию этой функции. Исходный код оригинальной версии функции поиска значения переменной представлен на рис. 4.



```
private static final Var lookup(Var vars, char name) {
    Var varIter = vars;
    while(varIter.name != name) {
        varIter = varIter.nextVar;
    }
    return varIter;
}
```

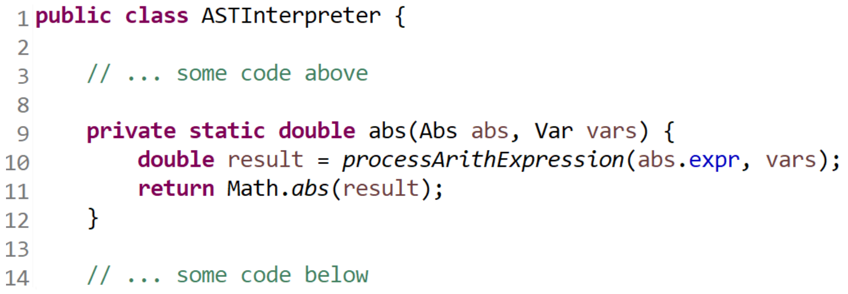
Рис. 4. Функция поиска значения переменной

Аргументами функции поиска являются таблица переменных и имя переменной, значение которой требуется найти. Таблица переменных в обоих исследуемых интерпретаторах представлена в виде списка пар – (имя_переменной, числовое_значение). Функция поиска содержит цикл, проходящий по указанному списку пар и сравнивающий имя переменной из таблицы с именем переменной, значение которой требуется найти.

Функция lookup имеет одинаковы код для обоих исследуемых интерпретаторов. Специализация функции lookup приводит к тому, что ее определение полностью отсутствует в специализированных версиях, а вместо вызова подставляется имя переменной в программе-результате, хранящей текущее значение переменной интерпретируемой арифметической программы (см. раздел 4).

5.4. Специализация вычисления модуля

В качестве примера специализации арифметического выражения выберем выражение $|x - d|$. На рис. 5 и рис. 6 представлен исходный код функции вычисляющих модуль. Результат специализации кода с рис.6 по выражению $|x - d|$ представлен на рис. 7.



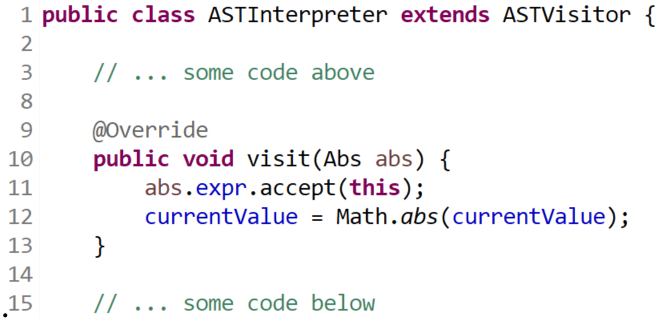
```java
public class ASTInterpreter {

    // ... some code above

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

    // ... some code below
```

Рис. 5. Вычисление модуля выражения рекурсивном спуске



```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
```

Рис. 6. Вычисление модуля выражения в шаблоне «посетитель»

```
 1 visit28: {
 2     accept30: {
 3         accept031: {
 4             visit30: {
 5                 accept32: {
 6                     accept033: {
 7                         visit32: {
 8                             obj40_currentValue = obj0_value;
 9                         }
10                     }
11                 }
12                 double left0 = obj40_currentValue;
13                 accept34: {
14                     accept035: {
15                         visit34: {
16                             obj40_currentValue = obj2_value;
17                         }
18                     }
19                 }
20                 double right = obj40_currentValue;
21                 obj40_currentValue = left0 - right;
22             }
23         }
24     }
25     obj40_currentValue = Math.abs(obj40_currentValue);
26 }
```

Рис. 7. Результат специализации выражения $|x-d|$

Строки 7–9 на рис. 7 соответствуют поиску значения переменной x в таблице значений, которое выполняется статически, т. е. код функции поиска отсутствует (см. предыдущий раздел 5.3). Значение переменной x интерпретируемой программы после специализации хранится в переменной obj0_value. Это значение копируется в переменную obj40_currentValue (строка 8). Скопированное значение x присваивается в переменную left0 (строка 12). В строках 15–17 осуществляется поиск значения переменной d, аналогично поиску значения x. В строке 20 значение d присваивается в переменную right. В строке 21 вычисляется значение $x-d$, а результат вычисления помещается в переменную obj40_currentValue.

Строки 2–24 на рис. 7 – это результат специализации строки 3 на рис. 6, т. е. эти строки должны содержать результат вычисления подвыражения внутри модуля $|x-d|$. Ровно такое подвыражение в строках 2–24 и вычисляется.

Строка 25 соответствует строке 4 на рис. 6. Как видно эти строки похожи. В них обращение к функции Math.abs является библиотечным вызовом, поэтому оно оставлено без изменений.

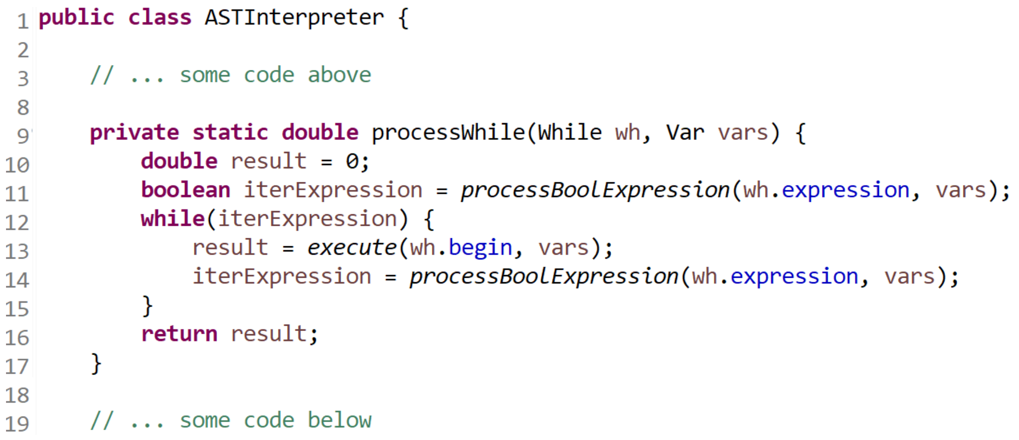
Специализированная версия функции вычисления модуля, полученная из интерпретатора, который работает методом рекурсивного спуска (рис. 5), очень похожа на рис. 7, поэтому мы не будем ее приводить. При желании с ней можно ознакомиться, перейдя по ссылке на github, которая содержит исходный и специализированный код интерпретаторов (ссылка находится в конце статьи).

5.5. Специализация цикла while

Рассмотрим обработку цикла while: версий до и после оптимизаций. Код интерпретаторов до оптимизаций представлен на рис. 8 и рис. 9.

Нужно отметить, что метод processWhile на рис. 8 содержит дополнительный аргумент vars, который представляет список пар (имя_переменной, числовое_значение) и является таблицей значений переменных в интерпретируемой программе. Этот список передается в виде аргумента, а не в виде глобальных переменных, чтобы увеличить силу оптимизаций, поскольку значения глобальных переменных считаются неизвестными в процессе специализации. Указанное ограничение связано не столько с подходом частичных вычислений, сколько с реализацией, и будет снято в будущем.

Также требуется отметить, что во фрагменте интерпретатора, построенного по шаблону «посетитель» (рис. 9), использованы поля-переменные currentValue и currentConditionValue. Эти поля содержатся в экземпляре интерпретатора, основанного на шаблоне «посетитель», и они нужны, чтобы возвращать вычисленные значения – конструкций. Так, после выполнения строки 4 (программы на рис. 9), переменная currentConditionValue содержит вычисленное логическое значение для условия цикла While, которое необходимо проверить перед первым вхождением в тело цикла. Аналогично, после выполнения строки 6, currentValue содержит результат однократного выполнения тела цикла.



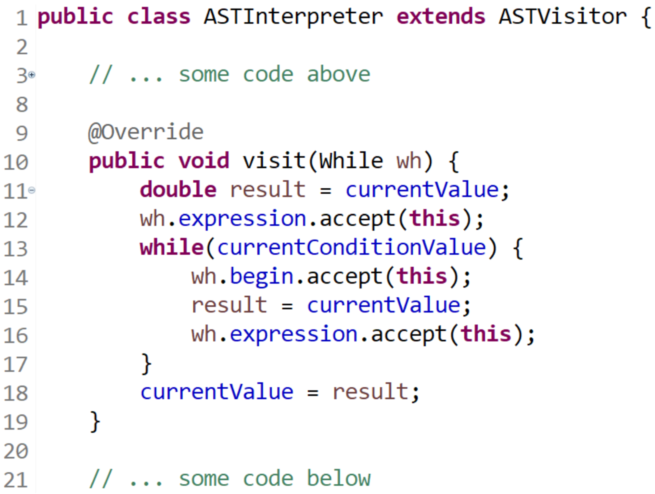
```
public class ASTInterpreter {

    // ... some code above

    private static double processWhile(While wh, Var vars) {
        double result = 0;
        boolean iterExpression = processBoolExpression(wh.expression, vars);
        while(iterExpression) {
            result = execute(wh.begin, vars);
            iterExpression = processBoolExpression(wh.expression, vars);
        }
        return result;
    }

    // ... some code below
```

Рис. 8. Обработка цикла while при рекурсивном спуске



```
public class ASTInterpreter extends ASTVisitor {

    // ... some code above

    @Override
    public void visit(While wh) {
        double result = currentValue;
        wh.expression.accept(this);
        while(currentConditionValue) {
            wh.begin.accept(this);
            result = currentValue;
            wh.expression.accept(this);
        }
        currentValue = result;
    }

    // ... some code below
```

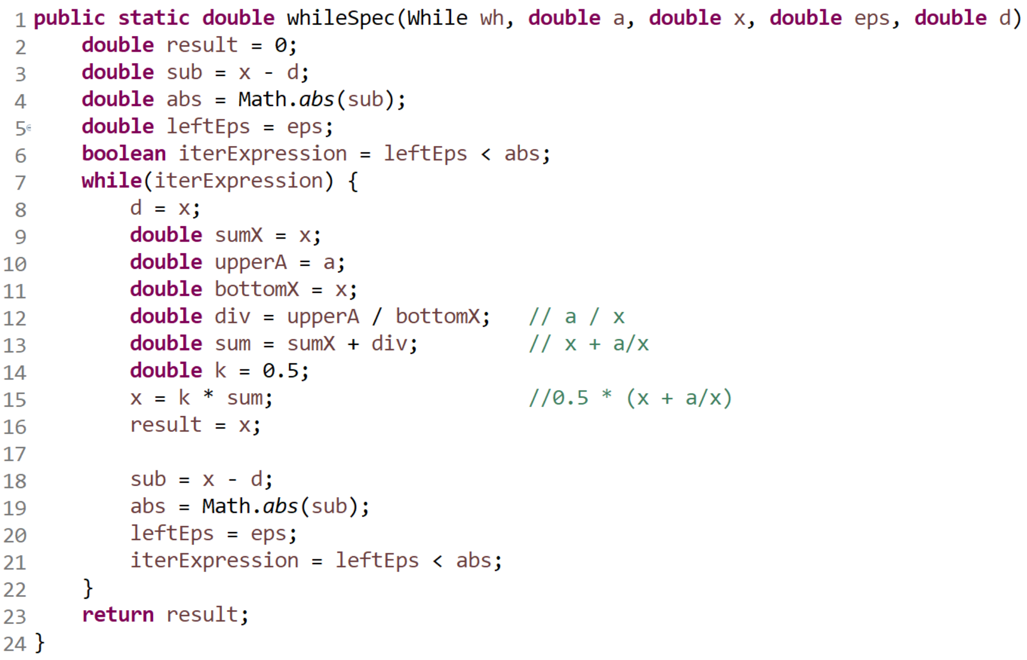
Рис.9. Обработка цикла while в шаблоне «посетитель»

На рис. 10 представлена схема результата специализации обеих версий интерпретатора (рекурсивным спуском и шаблоном «посетитель») и эта схема общая для обеих версий. В рамках данной статьи привести реальный код не представляется возможным, поскольку он занимает в сумме более 400 строк.

Нужно отметить, что специализатор JaSpe раскрыл вызовы обхода подвыражений и подставил их код в функцию, обрабатывающую While.

Результаты специализации интерпретаторов похожи и, поэтому имеют общую схему, в силу того что они специализируются по одной и той же программе, т. е. результат отражает структуру этой арифметической программы, общей для обоих случаев.

Прокомментируем результат специализации.



```
public static double whileSpec(While wh, double a, double x, double eps, double d)
    double result = 0;
    double sub = x - d;
    double abs = Math.abs(sub);
    double leftEps = eps;
    boolean iterExpression = leftEps < abs;
    while(iterExpression) {
        d = x;
        double sumX = x;
        double upperA = a;
        double bottomX = x;
        double div = upperA / bottomX;   // a / x
        double sum = sumX + div;         // x + a/x
        double k = 0.5;
        x = k * sum;                     //0.5 * (x + a/x)
        result = x;

        sub = x - d;
        abs = Math.abs(sub);
        leftEps = eps;
        iterExpression = leftEps < abs;
    }
    return result;
}
```

Рис. 10. Схема результата специализации цикла While по программе вычисления корня

В строке 3–6 производится вычисление условия цикла. Это вычисление разбито на вычисления отдельных подвыражений, поскольку интерпретаторы вычисляют их именно так. Специализатор JaSpe на данном этапе своего развития не собирает отдельные подвыражения в одно большое общее выражение и поэтому схема результата специализации цикла While содержит вычисление подвыражений, каждого в отдельной строке.

Итак, в строках 3–6 производится вычисление логического выражения условия цикла:

$$\varepsilon < |x - d|$$

Данное выражение вычисляется поэтапно, в строке 3 вычисляется $x - d$, в строке 4 модуль $|x - d|$, а в строке 6 находится результат итогового сравнения $\varepsilon < |x - d|$.

Условие цикла вычисляется также в строках 18–21, поскольку оно проверяется перед началом каждой итерации.

Так же поэтапно, как и условие цикла, вычисляется тело цикла. Тело цикла вычисляет значение выражения:

$$\frac{1}{2}\left(x + \frac{a}{x}\right)$$

В строке 12 вычисляется отношение $\frac{a}{x}$, в строке 13 – промежуточное $x + \frac{a}{x}$, а в строке 15 – итоговое выражение $\frac{1}{2}\left(x + \frac{a}{x}\right)$.

Несмотря на то, что результаты специализации интерпретаторов похожи, у их исходных версий есть одно существенное отличие. Интерпретатор на основе шаблона «посетитель» использует виртуальные вызовы, свойственные только объектно-ориентированному подходу к написанию программ. Анализ виртуальных вызовов представляет проблему для классических методов специализации. Наш частичный вычислитель JaSpe в ряде случаев умеет успешно оптимизировать виртуальные вызовы и выполнять динамическую диспетчеризацию во время анализа, получая результат близкий к уже изученной специализации императивных программ.

6. Измерение производительности

В экспериментах мы измеряли среднее время выполнения исходных и специализированных версий интерпретаторов в зависимости от точности, с которой необходимо извлечь квадратный корень. Число, чей квадратный корень вычисляется, было зафиксировано и равнялось 8. Само это число считалось неизвестным на этапе специализации и являлось параметром исследуемых программ.

Измерения проводились на компьютере с процессором Intel Core i5-3570 CPU @ 3.40GHz, объемом оперативной памяти 32 Гбайта и операционной системой Windows 10 Домашняя. В качестве Java-машины была выбрана Oracle Java JRE 16.0.1.

Непосредственно для самих измерений использовался инструмент Java Microbenchmark Harness (JMH) [30]. JMH позволяет выполнять нано/микро/мили/макро тесты производительности для Java программ, избегая при этом различных проблем измерений, которые создает Java-машина.

Каждое отдельное среднее время выполнения получалось по следующей процедуре:

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.

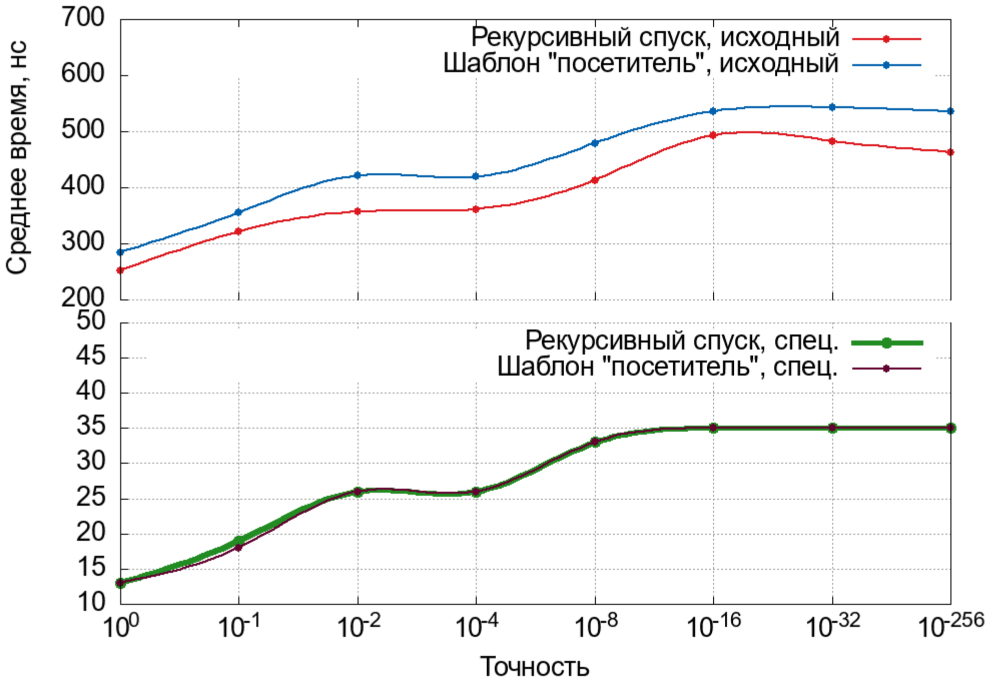


Рис. 11. Среднее время исполнения интерпретаторов, исходных и специализированных версий

Данные, полученные в соответствие с только что описанной процедурой измерений, изображены на рис. 11.

Относительная погрешность измерений для каждого из замеров не превосходила 3.3%. Погрешность вычислялась статистическими методами на основе предположения, что время исполнения интерпретаторов имеет нормальное распределение.

Ускорение интерпретатора, работающего методом рекурсивного спуска, составило от 12 до 19 раз. Для интерпретатора на основе шаблона «посетитель» ускорение находится в диапазоне от 14 до 22 раз.

7. Заключение

В статье исследуется специализация интерпретаторов, написанных в объектно-ориентированной парадигме. Рассматриваются два подхода к написанию интерпретаторов:

- метод рекурсивного спуска, пришедший из императивного программирования;
- шаблон «посетитель», характерный для объектно-ориентированного подхода.

Несмотря на то, что результаты специализации для обеих версий интерпретаторов достаточно близки, успешная специализация интерпретатора, написанного на основе шаблона «посетитель», требовала существенного развития метода частичных вычислений. Классические методы разрабатывались для функциональных и императивных программ и поэтому не справляются с полиморфизмом, свойственным объектно-ориентированной парадигме.

Выполненное нами в рамках работ [11, 12, 14, 15] развитие методов частичных вычислений, которое привело к созданию специализаторов CILPE и JaSpe, решает в ряде случаев проблему полиморфизма, с которой не справляются специализаторы-предшественники. Эти методы позволили на рассмотренной в статье задаче получить ускорение от 14 до 22 раз для интерпретатора, написанного в объектно-ориентированном стиле.

Настоящая работа основана на специализаторе JaSpe, который развивает методы, разработанные для специализатора CILPE, в следующих направлениях:

- JaSpe принимает и возвращает программы на языке высокого уровня Java, а не на внутреннем стековом языке SOOL (это свойство позволяет обычному программисту видеть и анализировать результаты специализации, без необходимости разбираться в деталях низкоуровневого языка);
- JaSpe ориентирован на интерактивное взаимодействие с программистом-пользователем (это свойство означает, что программисту легче понять, почему остаточная программа получилась той или иной, а также программисту проще управлять

процессом специализации для достижения требуемого ему результата).

Дальнейшее развитие методов частичных вычислений, разрабатываемых авторами настоящей статьи, планируется в направлении применения специализации к промышленным интерпретаторам. Для этого необходимо реализовать возможность специализации вызовов функций, чей исходный Java-код не доступен специализатору, но доступен байт-код этих функций. Например, такими вызовами являются вызовы библиотечных методов.

Исходный и специализированный код интерпретаторов доступен по адресу https://github.com/igor-adamovich/ArithInterpreters.

Литература

1. Jones N.D., Gomard C.K., Sestoft P. Partial Evaluation and Automatic Program Generation // Prentice-Hall, 1993. — http://www.itu.dk/~sestoft/pebook/pebook.html
2. Marlet R. Program Specialization // Wiley-ISTE, 2012. — 544 pp.
3. Turchin V.F. The Concept of a Supercompiler // ACM Transactions on Programming Languages and Systems, 1986, 8:3. — P. 292–325. — https://doi.org/10.1145/5956.5957
4. Turchin V.F. Supercompilation: Techniques and results // Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, 1996. — Lecture Notes in Computer Science, vol. 1181, Springer. — P. 227–248. — https://doi.org/10.1007/3-540-62064-8_20
5. Климов Анд.В. Введение в метавычисления и суперкомпиляцию // М.: КомКнига, 2008. — С. 343–368.
6. Romanenko S.A. Arity Raiser and its Use in Program Specialization. // Proceedings of the 3rd European Symposium on Programming (May 15 – 18, 1990). — Lecture Notes In Computer Science, vol. 432, Springer-Verlag, London. — P. 341–360. — https://doi.org/10.1007/3-540-52592-0_73
7. Futamura Y. Partial Evaluation of Computation Process — An Approach to a Compiler-Compiler // Systems, Computers, Controls. 1971, vol. 2, no. 5. — P. 45–50.
8. Futamura Y. EL1 Partial Evaluator (Progress Report) // Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, USA, 1973. — https://fi.ftmr.info/PE-Museum/EL1.PDF
9. Турчин В.Ф. и др. Базисный Рефал и его реализация на вычислительных машинах // М.: ЦНИПИАСС, 1977.
10. Адамович И.А., Климов Анд.В. Интерактивный специализатор подмножества языка Java, основанные на методе частичных

вычислений // Труды Института системного программирования РАН. 2018; 30(4). — С. 29–44. — https://doi.org/10.15514/ISPRAS-2018-30(4)-2

11. Адамович И.А., Климов Ю.А. Специализатор JaSpe: алгоритм внутрипроцедурного анализа времени связывания программ на подмножестве языка Java // Программные системы: теория и приложения, 2020, 11:1(44). — С. 3–29. — https://doi.org/10.25209/2079-3316-2020-11-1-3-29
12. Адамович И.А. Специализатор JaSpe: BT-объекты и межпроцедурный аспект алгоритма анализа времен связывания // Программные системы: теория и приложения, 2021, 12:4(51). — С. 3–33. — https://doi.org/10.25209/2079-3316-2021-12-4-3-32
13. Eclipse integrated development environment (IDE) / Eclipse Foundation. — https://www.eclipse.org/
14. Климов Ю.А. Возможности специализатора CILPE и примеры его применения к программам на объектно-ориентированных языках // Препринты ИПМ им.М.В.Келдыша, 2008, № 30. — 28 с. — https://keldysh.ru/papers/2008/source/prep2008_30.pdf
15. Климов Ю.А. Специализатор CILPE: частичные вычисления для объектно-ориентированных языков // Программные системы: теория и приложения, 2010, № 3 (3). — с. 13-36. — http://psta.psiras.ru/read/psta2010_3_13-36.pdf
16. Morris F.L. The next 700 formal language descriptions // Manuscript. Stanford University, California, USA, 1970.
17. Jones N.D., Sestoft P., Søndergaard H. Mix: A self-applicable partial evaluator for experiments in compiler generation // Lisp and Symbolic Computation, 1989, 2:9. — P. 9–50. — https://doi.org/10.1007/BF01806312
18. Романенко С.А. Генератор компиляторов, порожденный самоприменением специализатора, может иметь ясную и естественную структуру // М.:ИПМ им. М.В.Келдыша АН СССР, 1987, препринт N 26. — 35 стр. — https://pat.keldysh.ru/~roman/doc/1987-Romanenko--Generator_kompilyatorov__porozhdennyj_samoprimeneniem_specializatora--ru.pdf
19. Romanenko S.A. A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprisingly Natural and Understandable Structure // Partial Evaluation and Mixed Computation, North-Holland, 1988. — P. 445–463. — https://pat.keldysh.ru/~roman/doc/1988-Romanenko--A_Compiler_Generator_Produced_by_a_Self-Applicable_Specializer.pdf
20. Romanenko S.A. Arity Raiser and its Use in Program Specialization // Proceedings of the 3rd European Symposium on Programming. — Lecture Notes In Computer Science, vol. 432, 1990. — P. 341–360. https://doi.org/0.1007/3-540-52592-0_73

21. Consel C., Danvy O. Static and dynamic semantics processing // Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '91). — Association for Computing Machinery, New York, NY, USA, 1991. — P. 14–24. — https://doi.org/10.1145/99583.99588
22. Consel C., Khoo S.C. Semantics-directed generation of a Prolog compiler // Research Report 781, Yale University, New Haven, Connecticut, USA, 1990.
23. Würthinger T., Wimmer C., Wöß A., Stadler L., Duboscq G., Humer C., Richards G., Simon D., Wolczko M. One VM to rule them all // Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! New York, NY, USA, 2013. — P. 187–204. — https://doi.org/10.1145/2509578.2509581
24. Würthinger T., Wimmer C., Humer C., Wöß A., Stadler L., Seaton C., Duboscq G., Simon D., Grimmer M. Practical partial evaluation for high-performance dynamic language runtimes // SIGPLAN Not., 2017, vol. 52, no. 6. — P. 662–676. — https://doi.org/10.1145/3140587.3062381
25. Latifi F., Leopoldseder D., Wimmer C., Mössenbc̈k H. CompGen: generation of fast JIT compilers in a multi-language VM // Proc. of the 17th ACM SIGPLAN International Symposium on Dynamic Languages, DLS ’21, 2021. — P. 35–47. — https://doi.org/10.1145/3486602.3486930
26. Weisstein E.W. Newton's Iteration // MathWorld – A Wolfram Web Resource. — https://mathworld.wolfram.com/NewtonsIteration.html
27. Бахвалов Н.С., Кобельков Г.М., Жидков Н.П. Численные методы // М.: Лаборатория знаний, 2020. — 636 стр.
28. Aho A.V., Lam M.S., Sethi R., Ullman J. D. Compilers: Principles, Techniques, & Tools. 2nd edition // Addison Wesley, 2006. — 1040 pp.
29. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software // Addison Wesley, 1994. — 331 pp.
30. Java Microbenchmark Harness / Oracle Corporation. — https://github.com/openjdk/jmh

References

1. Jones N.D., Gomard C.K., Sestoft P. Partial Evaluation and Automatic Program Generation // Prentice-Hall, 1993. — http://www.itu.dk/~sestoft/pebook/pebook.html
2. Marlet R. Program Specialization // Wiley-ISTE, 2012. — 544 pp.
3. Turchin V.F. The Concept of a Supercompiler // ACM Transactions on Programming Languages and Systems, 1986, 8:3. — P. 292–325. — https://doi.org/10.1145/5956.5957
4. Turchin V.F. Supercompilation: Techniques and results // Perspectives of System Informatics, Second International Andrei Ershov Memorial

Conference, Akademgorodok, Novosibirsk, Russia, 1996. — Lecture Notes in Computer Science, vol. 1181, Springer. — P. 227–248. — https://doi.org/10.1007/3-540-62064-8_20

5. Klimov And.V. Vvedenie v metavychisleniya i superkompilyaciyu // M.: KomKniga, 2008. — C. 343–368.
6. Romanenko S.A. Arity Raiser and its Use in Program Specialization. // Proceedings of the 3rd European Symposium on Programming (May 15 – 18, 1990). — Lecture Notes In Computer Science, vol. 432, Springer-Verlag, London. — P. 341–360. — https://doi.org/10.1007/3-540-52592-0_73
7. Futamura Y. Partial Evaluation of Computation Process — An Approach to a Compiler-Compiler // Systems, Computers, Controls. 1971, vol. 2, no. 5. — P. 45–50.
8. Futamura Y. EL1 Partial Evaluator (Progress Report) // Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, USA, 1973. — https://fi.ftmr.info/PE-Museum/EL1.PDF
9. Turchin V.F. i dr. Bazisnyj Refal i ego realizaciya na vychislitel'nyh mashinah // M.: CNIPIASS, 1977.
10. Adamovich I.A., Klimov And.V. Interaktivnyj specializator podmnozhestva yazyka Java, osnovannye na metode chastichnyh vychislenij // Trudy Instituta sistemnogo programmirovaniya RAN. 2018; 30(4). — C. 29–44. — https://doi.org/10.15514/ISPRAS-2018-30(4)-2
11. Adamovich I.A., Klimov Yu.A. Specializator JaSpe: algoritm vnutriprocedurnogo analiza vremeni svyazyvaniya programm na podmnozhestve yazyka Java // Programmnye sistemy: teoriya i prilozheniya, 2020, 11:1(44). — C. 3–29. — https://doi.org/10.25209/2079-3316-2020-11-1-3-29
12. Adamovich I.A. Specializator JaSpe: BT-ob'ekty i mezhprocedurnyj aspekt algoritma analiza vremen svyazyvaniya // Programmnye sistemy: teoriya i prilozheniya, 2021, 12:4(51). — S. 3–33. — https://doi.org/10.25209/2079-3316-2021-12-4-3-32
13. Eclipse integrated development environment (IDE) / Eclipse Foundation. — https://www.eclipse.org/
14. Klimov Yu.A. Vozmozhnosti specializatora CILPE i primery ego primeneniya k programmam na ob'ektno-orientirovannyh yazykah // Preprinty IPM im.M.V.Keldysha, 2008, № 30. — 8 s. — https://keldysh.ru/papers/2008/source/prep2008_30.pdf
15. Klimov Yu.A. Specializator CILPE: chastichnye vychisleniya dlya ob"ektno-orientirovannyh yazykov // Programmnye sistemy: teoriya i prilozheniya, 2010, № 3 (3). — s. 13-36. — http://psta.psiras.ru/read/psta2010_3_13-36.pdf
16. Morris F.L. The next 700 formal language descriptions // Manuscript. Stanford University, California, USA, 1970.

17. Jones N.D., Sestoft P., Søndergaard H. Mix: A self-applicable partial evaluator for experiments in compiler generation // Lisp and Symbolic Computation, 1989, 2:9. — P. 9–50. — https://doi.org/10.1007/BF01806312
18. Romanenko S.A. Generator kompilyatorov, porozhdennyj samoprimeneniem specializatora, mozhet imet' yasnuyu i estestvennuyu strukturu // M.:IPM im. M.V.Keldysha AN SSSR, 1987, preprint N 26. — 35 str. — https://pat.keldysh.ru/~roman/doc/1987-Romanenko--Generator_kompilyatorov__porozhdennyj_samoprimeneniem_specializatora--ru.pdf
19. Romanenko S.A. A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprisingly Natural and Understandable Structure // Partial Evaluation and Mixed Computation, North-Holland, 1988. — P. 445–463. — https://pat.keldysh.ru/~roman/doc/1988-Romanenko--A_Compiler_Generator_Produced_by_a_Self-Applicable_Specializer.pdf
20. Romanenko S.A. Arity Raiser and its Use in Program Specialization // Proceedings of the 3rd European Symposium on Programming. — Lecture Notes In Computer Science, vol. 432, 1990. — P. 341–360. https://doi.org/0.1007/3-540-52592-0_73
21. Consel S., Danvy O. Static and dynamic semantics processing // Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '91). — Association for Computing Machinery, New York, NY, USA, 1991. — P. 14–24. — https://doi.org/10.1145/99583.99588
22. Consel C., Khoo S.C. Semantics-directed generation of a Prolog compiler // Research Report 781, Yale University, New Haven, Connecticut, USA, 1990.
23. Würthinger T., Wimmer C., Wöß A., Stadler L., Duboscq G., Humer C., Richards G., Simon D., Wolczko M. One VM to rule them all // Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! New York, NY, USA, 2013. — P. 187–204. — https://doi.org/10.1145/2509578.2509581
24. Würthinger T., Wimmer C., Humer C., Wöß A., Stadler L., Seaton C., Duboscq G., Simon D., Grimmer M. Practical partial evaluation for high-performance dynamic language runtimes // SIGPLAN Not., 2017, vol. 52, no. 6. — P. 662–676. — https://doi.org/10.1145/3140587.3062381
25. Latifi F., Leopoldseder D., Wimmer C., Mössenbčk H. CompGen: generation of fast JIT compilers in a multi-language VM // Proc. of the 17th ACM SIGPLAN International Symposium on Dynamic Languages, DLS ’21, 2021. — P. 35–47. — https://doi.org/10.1145/3486602.3486930
26. Weisstein E.W. Newton's Iteration // MathWorld – A Wolfram Web Resource. — https://mathworld.wolfram.com/NewtonsIteration.html

27. Bahvalov N.S., Kobel'kov G.M., Zhidkov N.P. Chislennye metody // M.:Laboratoriya znanij, 2020. — 636 str.
28. Aho A.V., Lam M.S., Sethi R., Ullman J. D. Compilers: Principles, Techniques, & Tools. 2nd edition // Addison Wesley, 2006. — 1040 pp.
29. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software // Addison Wesley, 1994. — 331 pp.
30. Java Microbenchmark Harness / Oracle Corporation. — https://github.com/openjdk/jmh