ISSN 2079-3316 **ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ** **т. 13**, № 4(55), с. 111–137
научная статья МАТЕМАТИЧЕСКИЕ ОСНОВЫ ПРОГРАММИРОВАНИЯ

УДК 519.681.3
10.25209/2079-3316-2022-13-4-111-137

Исследование эффективности специализации интерпретаторов на объектно-ориентированном языке Java методами частичных вычислений с BT-объектами

Игорь Алексеевич **Адамович**[1]✉, Юрий Андреевич **Климов**[2]

[1] Институт программных систем им. А. К. Айламазяна РАН, Веськово, Россия
[2] Институт прикладной математики им. М. В. Келдыша РАН, Москва, Россия

[1]✉*i.a.adamovich@gmail.com* *(подробнее об авторах на с. 133)*

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. *(see abstract in English on p. 134)*

Ключевые слова и фразы: интерпретаторы, компиляторы, частичные вычисления, специализация, метавычисления

Для цитирования: Адамович И. А., Климов Ю. А. *Исследование эффективности специализации интерпретаторов на объектно-ориентированном языке Java методами частичных вычислений с BT-объектами* // Программные системы: теория и приложения. 2022. **Т. 13**. № 4(55). С. 111–137. http://psta.psiras.ru/read/psta2022_4_111-137.pdf

Введение

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 эффективной компиляции программ через специализацию интерпретатора. Идеи компиляции программы с помощью специализации называются проекциями Футамуры–Турчина и были впервые предложены Есихико Футамурой [7, 8] и независимо открыты позже Валентином 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.

Авторы настоящей статьи считают, что развитие методов специализации программ и последующее их внедрение в практику создания программного обеспечения приведет к появлению новой продуктивной методологии разработки программ: изначально создается универсальная программа, которая в дальнейшем специализируются под конкретные условия применения. При этом эквивалентность универсальной и новой программ обеспечивается специализатором. Такая практика применения специализации позволяет, во-первых, повысить производительность труда (универсальная программа разрабатывается один раз для нескольких применений) и, во-вторых, обеспечить качество результата (специализация основана на эквивалентных преобразованиях).

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.

Специализатор JaSpe разрабатывается как интерактивный инструмент статического анализа и оптимизаций. Авторы настоящей статьи утверждают, что черно-ящичный подход к специализации достиг своих пределов. Для того, чтобы разрабатывать и применять более мощные инструменты, необходимо отойти от решения, в котором методы анализа и оптимизации программ работают в автоматическом режиме, без взаимодействия с человеком.

Методы специализации программ развиваются с 1970-х годов, но, к сожалению, до сих пор не находят практического применения. Мы считаем, что основные трудности связаны с тем, что задача специализации требуют гораздо большего участия человека для управления процессом специализации, анализа результатов, проведения компьютерных экспериментов. Данные идеи начинают поддерживать и другие авторы научных работ в области статического анализа программ, например [14, 15]

Также, необходимо отметить, что 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 [16, 17], 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 1 производится обзор области. В разделе 2 кратко описываются основные понятия и особенности частичных вычислений. 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 «Заключение» завершает данную статью и содержит основные выводы.

1. Обзор области

По нашим данным, первое построение компилятора из интерпретатора было получено Моррисом в 1970 году [**18**] ручным способом.

Первым из известных нам специализаторов, реализовавшим автоматическое порождение компиляторов, является MIX [**19**]. MIX специализировал программы на функциональном языке Mixwell. Работа [**19**] 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 [**20**, **21**] 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 [6]. 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 [22]. 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 [23]. Скомпилированные таким компилятором программы работали примерно в 6 раз быстрее чем интерпретируемые.

Современный частичный вычислитель, использующийся для компиляции методом специализации интерпретатора, встроен в фреймворк Truffle, компонент GraalVM [24,25]. 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 [26] 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 [16,17]. CILPE предназначен для специализации программ на языке SOOL: подмножестве языке CIL — промежуточного языка платформы Microsoft .NET.

В заключение необходимо упомянуть систему программирования Julia [27], поскольку она является одним из немногих примеров, когда специализацию программ удалось довести до практического программирования. В Julia специализация функций происходит не по

значениям некоторых аргументов функции, а по *типам* аргументов. Поэтому, чтобы достичь специализации, например, по конкретным числовым значениям, в Julia требуется «обертывать» константы в типы (т.е. превращать константы в типы). При этом, необходимо вносить изменения в текст программы, чтобы заменить работу с константами на работу с типами, их изображающими.

В случае «классического» специализатора, вроде JaSpe, исходная программа является «обыкновенной» константой, и специализация программы может быть выполнена без изменения ее текста (в отличие от системы Julia).

2. Частичные вычисления

В методе частичных вычислений обычно выделяют две стратегии online и offline. Эти стратегии в значительной степени отличаются друг от друга, поэтому сложно описать частичные вычисления в целом, без указания на ту или иную стратегию.

Специализация программ по online-стратегии предполагает один проход по оптимизируемой программе. При этом анализ и преобразование специализируемой программы выполняются одновременно. Как результат такого подхода, решение об оптимизации той или иной конструкции выполняется на основе неполной (локальной) информации о программе, т. к. специализатор не может использовать информацию в оставшейся, ещё не обработанной части программы. В процессе специализации частичный вычислитель, работающий по online-стратегии, оперирует с конкретными значениями переменных, а также абстрактным значением «неизвестно».

Частичный вычисления, работающие по offline-стратегии, выполняются в два этапа – анализ времен связывания (binding-time analysis) и генерация остаточной программы (residual program generation). На первом этапе, который называется анализ времен связывания, конструкции специализируемой программы разделяются на два типа – статические и динамические. Затем на этапе генерации остаточной программы статические конструкции будут исполнены, а динамические перейдут в результирующую (остаточную) программу.

Благодаря наличию двух проходов, на этапе генерации остаточной программы, специализатор может учитывать полную (глобальную) информацию о программе, собранную анализом времен связывания.

Настоящая работа основана на специализаторе JaSpe, который работает по offline-стратегии.

3. Первая проекция Футамуры

Рассмотрим одну из базовых идей, лежащих в основе компиляции методом специализации интерпретатора. 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 $SPEC(f_S(x,y),a)$ для программ $f_S(x,y)$ на языке S (здесь индексом будем обозначать язык, на котором написана данная программа). Предполагаем, что специализируемая программа $f_S(x,y)$ (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 $I_S(p_L(x),d)$, написанный на этом языке S, для программ $p_L(x)$ на языке L. Также, пусть дана некоторая программа $P_L(x)$ на языке L. Тогда, 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: $Q_S \stackrel{\text{def}}{=} SPEC(I_S,P_L)$. Эта программа Q_S функционально эквивалентна программе P_L. А именно, для любых исходных данных D, либо обе программы не завершаются, либо обе программы завершаются и $Q_S(D)=P_L(D)$, т.е. эти программы выдают одинаковый результат. Важно, что Q_S является программой, написанной на языке S, за счет этого ее выполнение более эффективно, чем если бы исходная программа P_L исполнялась универсальным интерпретатором I_S.

Итак, вместо программы P_L, написанной на языке L, получена программа Q_S на языке S, которая выполняет ту же функцию, что и P_L. Т. е. мы фактически скомпилировали программу P_L из языка L в язык S. Таким образом, первую проекцию Футамуры можно записать как:

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

4. Интерпретируемая программа

В нашей работе в качестве интерпретируемой программы выбрана программа вычисления квадратного корня с заданной точностью

методом Ньютона [28, 29]. Математическая формула, на которой основывается вычисление такова:

$$x_{i+1} = 1/2(x_i + a/x_i)$$

Псевдокод программы представлен на листинге 1.

Листинг 1. Вычисление квадратного корня методом Ньютона.

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

Переменные ϵ(точность) и a (подкоренное 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 [30], а описание шаблона «посетитель» в [31].

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 репозиторий с интерпретаторами, рассматриваемыми в данной работе, размещена 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. рисунок 1) – она определяет конкретный тип предложения и, основываясь на этом конкретном типе, вызывает соответствующую функцию обработки конкретной конструкции.

```
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
```

Рисунок 1. Функция processStatement интерпретатора, работающего рекурсивным спуском

Например для цикла «while» функцией обработки конкретной конструкции является

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

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-потомок ASTInterpreter. Код класса ASTVisitor, содержащий прототипы всех методов для посещения всех возможных типов узлов AST, помещен на рисунке 2.

```
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);
}
```

Рисунок 2. Объявление методов в классе ASTVisitor

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 рисунке 3 (accept в остальных классах, описывающих узлы AST, текстуально повторяет метод с рисунка 3). Примеры методов visitor.visit(...) можно найти ниже по ходу статьи (рисунки 6 и 9).

```
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
```

Рисунок 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», будет вызывать 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, 16, 17] 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 функции поиска значения переменной

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

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

Рисунок 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 вида (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 рисунках 5 и 6 представлен исходный код функции вычисляющих модуль.

```
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
```

Рисунок 5. Вычисление модуля выражения при рекурсивном спуске

```
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
```

Рисунок 6. Вычисление модуля выражения в шаблоне «посетитель»

Результат специализации кода с рисунка 6 по выражению $|x - d|$ представлен на рисунке 7.

Строки 6–8 на рисунке 7 соответствуют поиску значения переменной x в таблице значений, которое выполняется статически, т. е. код функции поиска отсутствует (см. предыдущий подраздел 5.3). Значение переменной x интерпретируемой программы после специализации хранится в переменной obj0_value. Это значение копируется в переменную obj40_currentValue (строка 7). Скопированное значение x присваивается в переменную left0 (строка 10). В строках 12–14 осуществляется поиск значения переменной d, аналогично поиску значения x. В строке 16 значение d присваивается в переменную right.

```
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 }
```

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

В строке 17 вычисляется значение $x - d$, а результат вычисления помещается в переменную obj40_currentValue.

Строки 3–19 на рисунке 7 – это результат специализации строки 3 на рисунке 6, т. е. эти строки должны содержать результат вычисления подвыражения внутри модуля $|x - d|$. Ровно такое подвыражение в строках 3–19 и вычисляется.

Строка 20 соответствует строке 4 на рисунке 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 (рисунок 5), очень похожа на рисунок 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 рисунках 8 и 9. Нужно отметить, что метод processWhile на рисунке 8 содержит

```
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
```

Рисунок 8. Обработка цикла while при рекурсивном спуске

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.

```
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
```

Рисунок 9. Обработка цикла while в шаблоне «посетитель»

Также требуется отметить, что во фрагменте интерпретатора, построенного по шаблону «посетитель» (рисунок 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 рисунке 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.

На рисунке 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.

```
1 public static double whileSpec(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;
13          double sum = sumX + div;
14          double k = 0.5;
15          x = k * sum;
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 }
```

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

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.

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:

$$\epsilon < |x - d|$$

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

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:

$$1/2(x + a/x)$$

В строке 12 вычисляется отношение a/x, в строке 13 – промежуточное $x + a/x$, а в строке 15 – итоговое выражение $1/2(x + a/x)$.

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) [**32**]. 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.

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

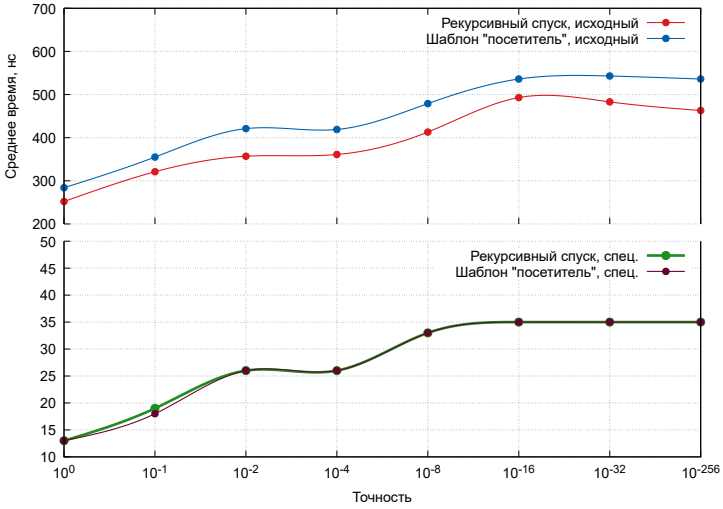

Рисунок 11. Среднее время исполнения интерпретаторов, исходных и специализированных версий

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**, **16**, **17**] 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 (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*.

Список литературы

[1] Jones N. D., Gomard C. K., Sestoft P. *Partial Evaluation and Automatic Program Generation*, Prentice-Hall International Series in Computer Science, 1st ed..– Prentice-Hall.– 1993.– ISBN 978-0130202499%.– 415 pp. ↑112

[2] Marlet R. *Program Specialization.*– Wiley–ISTE.– 2012.– ISBN 9781848213999.– 544 pp. ↑112

[3] Turchin V. F. *The concept of a supercompiler* // ACM Transactions on Programming Languages and Systems.– 1986.– Vol. **8**.– No. 3.– pp. 292–325. ↑112

[4] Turchin V. F. *Supercompilation: Techniques and results* // *Perspectives of System Informatics*, Second International Andrei Ershov Memorial

Conference (Akademgorodok, Novosibirsk, Russia, June 25–28, 1996), Lecture Notes in Computer Science.– vol. **1181**, eds. Bjørner D., Broy M., Pottosin I. V., Berlin–Heidelberg: Springer.– ISBN 978-3-540-49637-3.– pp. 227–248. ↑112

[5] Климов Анд. В. *Введение в метавычисления и суперкомпиляцию* // *Будущее прикладной математики: Лекции для молодых исследователей. От идей к технологиям*, М.: КомКнига.– 2008.– ISBN 978-5-484-01028-8.– с. 343–368. ↑112

[6] 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. **482**, Berlin–Heidelberg: Springer.– 1990.– ISBN 978-3-540-47045-8.– pp. 341–360. ↑112, 115

[7] Futamura Y. *Partial evaluation of computation process — an approach to a compiler-compiler* // Higher-Order and Symbolic Computation.– 1999.– Vol. **12**.– No. 4.– pp. 381–391. ↑112, 117

[8] Futamura Y. *EL1 Partial Evaluator*, Progress Report.– Cambridge, Massachusetts, USA: Center for Research in Computing Technology, Harvard University.– 1973.– 12 pp. ↑112, 117

[9] Турчин И. Ф. и др. *Базисный РЕФАЛ и его реализация на вычислительных машинах*, Фонд алгоритмов и программ для ЭВМ (в отрасли «Строительство»).– Т. **40**.– М.: ЦНИПИАСС.– 1977.– 263 с. ↑112, 117

[10] Адамович И. А., Климов Анд. В. *Интерактивный специализатор подмножества языка Java, основанный на методе частичных вычислений* // Труды Института системного программирования РАН.– 2018.– Т. **30**.– № 4.– с. 29–44 (In English). ↑113

[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.– Т. **11**.– № 1(44).– с. 3–29. ↑113, 114, 121, 129

[12] Адамович И. А. *Специализатор JaSpe: BT-объекты и межпроцедурный аспект алгоритма анализа времен связывания* // Программные системы: теория и приложения.– 2021.– Т. **12**.– № 4(51).– с. 3–33. ↑113, 114, 121, 129

[13] *Eclipse integrated develoment environment (IDE)*.– Eclipse Foundation. ↑113

[14] Nadeem A. *Human-centered approach to static-analysis-driven developer tools* // Communications of the ACM.– 2022.– Vol. **65**.– No. 3.– pp. 38–45. ↑113

[15] Tiganov D., Do L. N. Q., Ali K. *Designing UIs for static-analysis tools* // Communications of the ACM.– 2022.– Vol. **65**.– No. 2.– pp. 52–58. ↑113

[16] Климов Ю. А. *Возможности специализатора CILPE и примеры его применения к программам на объектно-ориентированных языках* // Препринты ИПМ им. М. В. Келдыша.– 2008.– 030.– 28 с. ↑113, 115, 121, 129

[17] Климов Ю. А. *Специализатор CILPE: частичные вычисления для*

объектно-ориентированных языков // Программные системы: теория и приложения.– 2010.– Т. **1**.– № 3(3).– с. 13–36. URL ↑113, 115, 121, 129

[18] Morris F. L. *The next 700 formal language descriptions* // LISP and Symbolic Computation.– November 1993.– Vol. **6**.– No. 3–4.– pp. 249–257. doi ↑114

[19] Jones N. D., Sestoft P., Søndergaard H. *Mix: A self-applicable partial evaluator for experiments in compiler generation* // LISP and Symbolic Computation.– Febrary 1989.– Vol. **2**.– No. 9.– pp. 9–50. doi ↑114

[20] Романенко С. А. *Генератор компиляторов, порожденный самоприменением специализатора, может иметь ясную и естественную структуру* // Препринты ИПМ им. М. В. Келдыша.– 1987.– 026.– 39 с. URL ↑114

[21] Romanenko S. A. *Compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure* // *Partial Evaluation and Mixed Computation*, North–Holland: Elsevier Sxience Publishers B.V..– 1988.– с. 445–463. URL ↑114

[22] Consel C., Danvy O. *Static and dynamic semantics processing* // *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (January 21–23, 1991, Orlando, Florida, USA), New York: ACM.– 1991.– ISBN 978-0-89791-419-2.– pp. 14–24. doi ↑115

[23] Consel C., Khoo S. C. *Semantics-directed generation of a prolog compile* // Science of Computer Programming.– 1993.– Vol. **21**.– No. 3.– pp. 263–291. doi ↑115

[24] 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* (October29–31, 2013, Indianapolis, Indiana, USA), New York: ACM.– 2013.– ISBN 978-1-4503-2472-4.– с. 187–204. doi ↑115

[25] 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* // ACM SIGPLAN Notices.– June 2017.– Т. **52**.– № 6.– с. 662–676. doi ↑115

[26] Latifi F., Leopoldseder D., Wimmer C., Mössenbčk H. *CompGen: generation of fast JIT compilers in a multi-language VM* // *Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2021 (19 October 2021, Chicago, IL, USA).– 2021.– ISBN 978-1-4503-9105-4.– с. 35–47. doi ↑115

[27] Bezanson J., Edelman A., Karpinski S., Shah V. B. *Julia: A Fresh Approach to Numerical Computing* // SIAM Review.– 2017.– Т. **59**.– № 1.– с. 65–98. doi ↑115

[28] Weisstein E. W. *Newton's Iteration.*– MathWorld – A Wolfram Web Resource. URL ↑118

[29] Бахвалов Н. С., Кобельков Г. М., Жидков Н. П. *Численные методы.*– М.: Лаборатория знаний.– 2020.– ISBN 978-5-00101-836-0.– 636 с. ↑118

[30] Aho A. V., Lam M. S., Sethi R., Ullman J. D. *Compilers: Principles, Techniques, and Tools*, 2nd ed..– Addison Wesley.– 2006.– ISBN 978-0321486813.– 1040 с. ↑118

[31] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed..– Addison Wesley.– 1994.– ISBN 978-0201633610.– 331 с. ↑118

[32] *Java Microbenchmark Harness.*– Oracle Corporation. URL ↑128

Поступила в редакцию 02.11.2022;
одобрена после рецензирования 05.11.2022;
принята к публикации 05.12.2022.

Рекомендовал к публикации *к.ф.-м.н. С. А. Романенко*

Информация об авторах:

Игорь Алексеевич Адамович

Научный сотрудник Института программных систем имени А. К. Айламазяна РАН. Научные интересы: метавычисления, суперкомпиляция, частичные вычисления, верификация программ, разработка на ПЛИС и ASIC. Принимал активное участие в разработке коммуникационных сетей суперкомпьютера «СКИФ-Аврора» «Паутина» и «3D-тор»

0000-0001-9728-3024
e-mail: *i.a.adamovich@gmail.com*

Юрий Андреевич Климов

Старший научный сотрудник ИПМ им. М.В. Келдыша РАН, к.ф.-м.н. Разработчик метода специализации на основе частичных вычислений, принимал активное участие в разработке коммуникационного ПО для сетей SCI, 3D-тор суперкомпьютера «СКИФ-Аврора» и «МВС-Экспресс» суперкомпьютера К-100.

0000-0001-5081-1547
e-mail: *yuri@klimov.net*

Вклад авторов: *И. А. Адамович* – 70% (идея, методология, программное обеспечение, расследование, сбор материала, написание черновой версии, доработка и редактирование, визуализация, администрирование); *Ю. А. Климов* – 30% (идея, методология, курирование данных, доработка и редактирование).

Авторы заявляют об отсутствии конфликта интересов.

ISSN 2079-3316 **PROGRAM SYSTEMS: THEORY AND APPLICATIONS** **vol. 13**, No 4(55), pp. 111–137
Research Article THEORETICAL COMPUTER SCIENCE

UDC 519.681.3
10.25209/2079-3316-2022-13-4-111-137

Efficiency Investigation of BT-Object Based Partial Evaluation of Interpreters Written in the Java Object-Oriented Language

Igor Alekseevich **Adamovich**[1], Yuri Andreevich **Klimov**[2]

[1] Ailamazyan Program Systems Institute of RAS, Ves'kovo, Russia
[2] Keldysh Institute of Applied Mathematics of RAS, Moscow, Russia

[1]✉*i.a.adamovich@gmail.com* *(learn more about the authors in Russian on p. 133)*

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. (*In Russian*).

Key words and phrases: interpreters, compilers, partial evaluation, specialization, metacomputations

2020 *Mathematics Subject Classification*: 68N15; 68N19, 68N20

For citation: Igor A. Adamovich, Yuri A. Klimov. *Efficiency Investigation of BT-Object Based Partial Evaluation of Interpreters Written in the Java Object-Oriented Language* // Program Systems: Theory and Applications, 2022, **13**:4(55), pp. 111–137. (*In Russian*). http://psta.psiras.ru/read/psta2022_4_111-137.pdf

References

[1] N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*, Prentice-Hall International Series in Computer

Science, 1st ed., Prentice-Hall, 1993, ISBN 978-0130202499%, 415 pp. ↑112

[2] R. Marlet. *Program Specialization*, Wiley–ISTE, 2012, ISBN 9781848213999, 544 pp. ↑112

[3] V. F. Turchin. "The concept of a supercompiler", *ACM Transactions on Programming Languages and Systems*, **8**:3 (1986), pp. 292–325. ↑112

[4] V. F. Turchin. "Supercompilation: Techniques and results", *Perspectives of System Informatics*, Second International Andrei Ershov Memorial Conference (Akademgorodok, Novosibirsk, Russia, June 25–28, 1996), Lecture Notes in Computer Science, vol. **1181**, eds. Bjørner D., Broy M., Pottosin I. V., Springer, Berlin–Heidelberg, ISBN 978-3-540-49637-3, pp. 227–248. ↑112

[5] And. V. Klimov. "Introduction to metacomputations and supercompilation", *Budushcheye prikladnoy matematiki: Lektsii dlya molodykh issledovateley. Ot idey k tekhnologiyam*, KomKniga, M., 2008, ISBN 978-5-484-01028-8, pp. 343–368 (In Russian). ↑112

[6] S. A. Romanenko. "Arity raiser and its use in program specialization", *Proceedings of the 3rd European Symposium on Programming*, Lecture Notes in Computer Science, vol. **482**, Springer, Berlin–Heidelberg, 1990, ISBN 978-3-540-47045-8, pp. 341–360. ↑112, 115

[7] Y. Futamura. "Partial evaluation of computation process — an approach to a compiler-compiler", *Higher-Order and Symbolic Computation*, **12**:4 (1999), pp. 381–391. ↑112, 117

[8] Y. Futamura. *EL1 Partial Evaluator*, Progress Report, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, USA, 1973, 12 pp. ↑112, 117

[9] I. F. i dr. Turchin. *Basic Refal and its Implementation on Computers*, Fond algoritmov i programm dlya EVM (v otrasli "Stroitel'stvo"), vol. **40**, TsNIPIASS, M., 1977 (In Russian), 263 pp. ↑112, 117

[10] I. A. Adamovich, And. V. Klimov. "An interactive specializer based on partial evaluation for a Java subset", *Trudy Instituta sistemnogo programmirovaniya RAN*, **30**:4 (2018), pp. 29–44 (In English). ↑113

[11] I. A. Adamovich, Yu. A. Klimov. "The JaSpe specializer: an algorithm of intra-procedural binding time analysis in Java language subset", *Program Systems: Theory and Applications*, **11**:1(44) (2020), pp. 3–29 (In Russian). ↑113, 114, 121, 129

[12] I. A. Adamovich. "The JaSpe specializer: an algorithm of intra-procedural binding time analysis in Java language subset", *Program Systems: Theory and Applications*, **12**:4(51) (2021), pp. 3–33 (In Russian). ↑113, 114, 121, 129

[13] *Eclipse integrated develoment environment (IDE)*, Eclipse Foundation. URL ↑113

[14] A. Nadeem. "Human-centered approach to static-analysis-driven developer tools", *Communications of the ACM*, **65**:3 (2022), pp. 38–45. doi ↑113

[15] D. Tiganov, L. N. Q. Do, K. Ali. "Designing UIs for static-analysis tools", *Communications of the ACM*, **65**:2 (2022), pp. 52–58. doi ↑113

[16] Yu. A. Klimov. "Specializer CILPE: examples of object-oriented program specialization", *Preprinty IPM im. M. V. Keldysha*, 2008, 030 (In Russian), 28 pp. URL ↑113, 115, 121, 129

[17] Yu. A. Klimov. "Specializer CILPE: partial evaluator for object-oriented languages", *Program Systems: Theory and Applications*, **1**:3(3) (2010), pp. 13–36 (In Russian). URL ↑113, 115, 121, 129

[18] F. L. Morris. "The next 700 formal language descriptions", *LISP and Symbolic Computation*, **6**:3–4 (November 1993), pp. 249–257. doi ↑114

[19] N. D. Jones, P. Sestoft, H. Søndergaard. "Mix: A self-applicable partial evaluator for experiments in compiler generation", *LISP and Symbolic Computation*, **2**:9 (Febrary 1989), pp. 9–50. doi ↑114

[20] S. A. Romanenko. "A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure", *Preprinty IPM im. M. V. Keldysha*, 1987, 026 (In Russian), 39 pp. URL ↑114

[21] S. A. Romanenko. "Compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure", *Partial Evaluation and Mixed Computation*, Elsevier Sxience Publishers B.V., North–Holland, 1988, pp. 445–463. URL ↑114

[22] S. Consel, O. Danvy. "Static and dynamic semantics processing", *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (January 21–23, 1991, Orlando, Florida, USA), ACM, New York, 1991, ISBN 978-0-89791-419-2, pp. 14–24. doi ↑115

[23] C. Consel, S. C. Khoo. "Semantics-directed generation of a prolog compile", *Science of Computer Programming*, **21**:3 (1993), pp. 263–291. doi ↑115

[24] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, M. Wolczko. "One VM to rule them all", *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (October29–31, 2013, Indianapolis, Indiana, USA), ACM, New York, 2013, ISBN 978-1-4503-2472-4, pp. 187–204. doi ↑115

[25] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, M. Grimmer. "Practical partial evaluation for high-performance dynamic language runtimes", *ACM SIGPLAN Notices*, **52**:6 (June 2017), pp. 662–676. doi ↑115

[26] F. Latifi, D. Leopoldseder, C. Wimmer, H. Mössenbčk. "CompGen: generation of fast JIT compilers in a multi-language VM", *Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2021 (19 October 2021, Chicago, IL, USA), 2021, ISBN 978-1-4503-9105-4, pp. 35–47. doi ↑115

[27] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah. "Julia: A Fresh Approach to Numerical Computing", *SIAM Review*, **59**:1 (2017), pp. 65–98. doi ↑115

[28] E. W. Weisstein. *Newton's Iteration*, MathWorld — A Wolfram Web Resource. URL ↑118

[29] N. S. Bakhvalov, G. M. Kobel'kov, Zhidkov N. P.. *Numerical Methods*, Laboratoriya znaniy, M., 2020, ISBN 978-5-00101-836-0 (In Russian), 636 pp. ↑118

[30] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, 2nd ed., Addison Wesley, 2006, ISBN 978-0321486813, 1040 pp. ↑118

[31] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed., Addison Wesley, 1994, ISBN 978-0201633610, 331 pp. ↑118

[32] *Java Microbenchmark Harness*, Oracle Corporation. URL ↑128