ISSN 2079-3316 **ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ** **т. 11**, № 1(44), с. 3–29

ББК Э973.2
ГРНТИ 50.05.09, 50.41.17
УДК 519.681.3

И. А. Адамович, Ю. А. Климов

Специализатор JaSpe: алгоритм внутрипроцедурного анализа времени связывания программ на подмножестве языка Java

Аннотация. Анализ времени связывания в частичных вычислениях, нацеленных на оптимизацию программ, разделяет программные конструкции на статические и динамические. Статические конструкции исполняются специализатором, а динамические переходят в результирующую программу. Частичные вычисления применяются в основном для нетривиальной компиляции программ без компилятора, при наличии лишь интерпретатора и специализатора. Эффективность их существенно зависит от качества разметки программы, получаемой в результате анализа времени связывания.

Статья посвящена особенностям алгоритма анализа времени связывания специализатора JaSpe, разрабатываемого авторами данной публикации для широко распространенного объектно-ориентированного языка Java. Она содержит основные понятия, использованные при реализации анализа времени связывания, внутрипроцедурную версию алгоритма и обсуждение деталей анализа конструкций, использующих ссылочные типы данных.

Алгоритм отличается от предшествующих аналогов, из числа работающих с программами на объектно-ориентированных языках, нетривиальной обработкой конструкций ветвления (if, switch), циклов (for, while, do) и блочных инструкции, которые содержат последовательность других инструкций. От аналогичных алгоритмов, работающих с императивными и функциональными языками, он отличается использованием BT-объектов, которые позволяют получать более точную разметку — с большей долей статических конструкций — при обработке объектно-ориентированных программ. Алгоритм ориентирован на интерактивность и удобочитаемость результатов.

Ключевые слова и фразы: современные языки программирования, статический анализ программ, преобразование программ, метавычисления, смешанные вычисления, интерактивная специализация.

Исследование выполнено при финансовой поддержке РФФИ автора[1], проект № 18-37-00454.

Введение

Специализация — это оптимизация программ на основе использования априорной информации о значении части переменных. Пусть дана программа $f(x, y)$ от двух аргументов x и y и значение одного из ее аргументов $x = a$. Результатом специализации программы $f(x, y)$ по известному аргументу $x = a$ называется новая программа $g(y)$, зависящая от одного аргумента и обладающая следующим свойством: $f(a, y) = g(y)$ для любого y.

Важно заметить, что специализировать программу $f(x, y)$ возможно даже в том случае, если все ее аргументы неизвестны. При этом оптимизация будет производиться на основе информации, содержащейся в самой программе и известной на стадии оптимизации. Такой информацией могут быть числовые и другие константы или знание конкретных ссылочных типов, задействованных в той или иной конструкции.

Важными применениями специализации, помимо оптимизации программ, также могут быть компиляция программ без компилятора, при наличии лишь интерпретатора, и генерация компилятора из интерпретатора [1].

Одним из методов специализации являются частичные вычисления. Этот метод состоит из двух основных частей: *анализа времени связывания* и *генерации остаточной программы*.

Анализ времени связывания (Binding Time Analysis, BTA) — это первый этап специализации методом частичных вычислений. На этом этапе происходит разметка инструкций (statements), выражений и других программных конструкций. В классической версии BTA [2] конструкции языка размечаются как *статические* или *динамические*. Статические конструкции будут исполнены на этапе специализации, а динамические перейдут в результирующую (остаточную) программу. Второй этап специализации методом частичных вычислений, который отвечает за исполнение статических конструкций и переход в остаточную программу динамических, называется *генерацией остаточной программы* (Residual Program Generation, RPG). На втором этапе специализации и происходят *частичные вычисления*.

Одна и та же программа может потенциально иметь много разных разметок. Цель анализа времени связывания — построить такую разметку, при которой как можно больше конструкций размечены как статические, что приведёт к исполнению этих конструкций на

стадии специализации, а значит остаточная программа получится более эффективной.

Настоящая статья посвящена особенностям алгоритма анализа времени связывания специализатора JaSpe, разрабатываемого авторами данной публикации для широко распространенного объектно-ориентированного языка Java. В ней мы обсуждаем внутрипроцедурную (intra-procedural) часть алгоритма, работающего с подмножеством Java 8, которое включает в себя: условные инструкции `if` и `switch`; циклы `for`, `while`, `do-while`; блочные инструкции; присваивания; инструкция `return`; определения переменных, массивов, аргументов и полей; лексические литералы; выражения с унарным, бинарным операторами; выражения создания экземпляра класса; выражения доступа к полям объектов, к элементам в массиве; выражение `instanceof`. Конструкциями, которые не входят в анализируемое подмножество являются объявления локальных классов, цикл `for-each`, инструкции `continue` и `break`, а также конструкции, связанные с лямбда-выражениями, исключениями, многопоточностью, рефлексией и вызовами методов. Помимо этого, не рассматриваются программы, в которых явно заданы (определены или переопределены) конструкторы классов.

Первая версия алгоритма BTA, реализованная в начальной версии специализатора JaSpe, оптимизировала выражения примитивного типа [3], в то время как в данной статье описывается алгоритм BTA, который работает как с примитивными, так и со ссылочными типами.

Перед дальнейшим изложением необходимо сделать три замечания. Во-первых, не следует путать такие понятия анализа времени связывания, как «статические конструкции (инструкции, выражения)», с понятиями «статические методы и классы» (static), используемые в языке Java. Во-вторых, чтобы избежать путаницы, вместо терминов *статический* и *динамический* будем использовать символы S (статический) и D (динамический), например S-разметка (статическая разметка), D-разметка (динамическая разметка). В-третьих, S и D будут также использоваться для обозначения базовых вариантов разметки (мы будем называть такие базовые типы разметки «примитивными»). Эти варианты разметки используются для переменных, выражений и других конструкций, имеющих примитивный тип в языке Java. Для конструкций, имеющих ссылочный тип в Java, используются составные BT-объекты, которые основываются на базовых вариантах разметки. О BT-объектах будет подробнее рассказано ниже в подразделах 1.1 и 1.2.

В работе [4] приводятся свойства, классифицирующие варианты реализации алгоритмов BT-анализа. В соответствии с этой классификацией реализованный в JaSpe алгоритм внутрипроцедурного анализа времени связывания является:

- поливариантным по переменным — в разных частях программы специализатор может разметить одну и ту же переменную по-разному. Это позволяет более эффективно производить специализацию кода, в котором одна и та же переменная используется в разных местах программы с разными целями.
- поливариантным по классам — каждый класс может обладать несколькими разметками. Это позволяет по-разному разметить различные использования объектов данного класса.
- моновариантным по операциям — каждая операция размечается только одним способом независимо от пути вычисления, по которому можно прийти к данной операции.

Вклад данной работы следующий: приводится описание алгоритма внутрипроцедурного анализа времени связывания для объектно-ориентированного языка. Этот алгоритм поддерживает более широкое множество конструкций языка, чем предшествующие алгоритмы BTA для объектно-ориентированных языков (см. сопоставление в разделе 5).

Алгоритм анализа времени связывания, о котором идёт речь в данной статье, основывается на некоторых принципах, реализованных в специализаторе CILPE [4–11]. В тех фрагментах статьи, где описываемые принципы существенно отличаются от принципов, на которых основан специализатор CILPE, будет приводится сравнение этих принципов.

Последующее изложение материала структурировано следующим образом. В разделе 1 обсуждаются понятия, связанные с разметкой программы. В разделе 2 описан алгоритм разметки программ из подмножества языка Java. В разделе 3 обсуждается моновариантность BT-объектов и связанную с ней необходимость переразмечать программу. В разделе 4 описан алгоритм слияния BT-объектов и этот алгоритм иллюстрируется примерами. В разделе 5 производится обзор области, к которой принадлежит эта статья.

1. Разметка программы

Разметка программы состоит из двух составляющих: разметки задействованных в специализации классов, а также BT-кучи. Каждая

разметка класса описывается именем класса и разметкой методов этого класса. В свою очередь, разметка метода состоит из имени метода, списка BT-значений и BT-объектов, обозначающих разметку аргументов метода, и разметки тела метода. Одному аргументу метода соответствует одно BT-значение или один BT-объект, в зависимости от того, примитивного или ссылочного типа аргумент содержится в исходной программе.

Разметка тела метода состоит из разметки инструкций. Каждая инструкция может быть размечена одним из двух примитивных BT-значений — S или D. Если инструкция имеет BT-значение D, то она перейдёт в остаточную программу. Если инструкция имеет BT-значение S, то она будет выполнена на этапе генерации остаточной программы. Исключение составляют инструкция определения переменной. Инструкция определения переменной, имея S-разметку, может остаться в остаточной программе, если эта переменная получает D-разметку на каком-то другом участке программы (излагаемый алгоритм BTA поливариантен по переменным, что означает, что переменная может иметь разную BT-разметку на разных участках программы).

1.1. BT-куча

В работах [4–11] введено понятие *BT-кучи*: «BT-куча описывает все возможные состояния динамической памяти (кучи) во время исполнения программы. Она показывает, какая часть объектов статическая, т. е. над какими объектами операции могут быть выполнены во время генерации остаточной программы, а какая часть динамическая — операции над этими объектами перейдут в остаточную программу. Такие объекты будут созданы и операции над ними будут выполнены во время исполнения остаточной программы.»

В данной публикации используется формальное определение BT-кучи, модифицированное по сравнению с [4, 5]: BT-куча — это совокупность всех *BT-объектов*, созданных при анализе той или иной программы. BT-объект — это тройка:

- примитивное BT-значение S или D;
- список ссылочных типов (классов);
- отображение имен полей всех классов, определенных в этом списке типов, в примитивное BT-значение или в другой BT-объект — в зависимости от того, какой Java-тип имеет соответствующее поле — примитивный или ссылочный.

Важно отметить, что ВТ-куча может содержать несколько ВТ-объектов, хранящих в списке типов один и тот же класс. Это означает, что некоторые ВТ-объекты могут размечать один и тот же класс и эти разметки могут отличаться. Следовательно, рассматриваемый алгоритм ВТА является поливариантным по классам.

1.2. ВТ-объекты

Понятие ВТ-объекта формально введено в предыдущем разделе. Каждый ВТ-объект — это вариант разметки класса. Одному классу могут соответствовать несколько ВТ-объектов (ВТА поливариантный по классам). ВТ-объект актуален только в процессе специализации. Можно провести аналогию: в процессе исполнения программы порождаются экземпляры класса — объекты, а в процессе абстрактной интерпретации ВТА — ВТ-объекты.

Разметка ВТ-объекта является тройкой, первый компонент которой — это разметка верхнего уровня для данного ВТ-объекта. Она бывает только двух типов, S или D. В случае ВТ-объекта с S-разметкой верхнего уровня (*S-ВТ-объекта*) экземпляры классов, которым соответствует данный ВТ-объект, могут быть «разделены» на поля. В момент исполнения программы экземпляры класса, соответствующие S-ВТ-объекту, могут не создаваться. Также, если ВТ-объект обладает S-разметкой верхнего уровня, то это означает, что специализатору известен конкретный тип тех экземпляров класса, которым соответствует данный ВТ-объект. ВТ-объект с D-разметкой верхнего уровня (*D-ВТ-объект*) содержит только D-поля. Это значит, что все примитивные поля D-ВТ-объекта имеют разметку D и не могут быть удалены в остаточной программе. Ссылочные поля D-ВТ-объекта также будут являться D-ВТ-объектами. Конкретный тип экземпляров класса, соответствующих D-ВТ-объекту, в процессе специализации считается неизвестным.

Второй компонент тройки в определении ВТ-объекта — список типов — содержит типы, чью разметку описывает данный ВТ-объект.

И, наконец, третья, основная часть ВТ-объекта состоит из пар, образуемых квалифицированным именем поля и его разметкой. В случае, если поле имеет примитивный тип, то разметка может быть только S или D. В случае поля ссылочного типа, разметке поля соответствует ссылка на ВТ-объект.

2. Алгоритм анализа времени связывания

Реализованный в JaSpe алгоритм BTA основан на абстрактной интерпретации. При старте BTA начинает исполнять правила анализа, имитирующие обычное исполнение программы. Анализ программы начинается с метода, отмеченного как *точка входа в специализацию* (specialization entry point). На данном этапе развития алгоритма, точка входа в специализацию должна иметь в своём определении квалификаторы `final` и `static`. Квалификатор `final` означает, что данный метод не может быть переопределён в классах-потомках, а квалификатор `static` означает, что метод классовый (т. е. присущ классу, а не его экземплярам).

Аргументы точки входа и все статические поля всех классов получают D-разметку. Затем, BTA последовательно «абстрактно интерпретирует» инструкции точки входа начиная с первой. Изначально каждой инструкции присваивается BT-значение S. После чего осуществляется обход вложенных инструкций и подвыражений. На основе BT-значений вложенных инструкций и подвыражений значение головной инструкции может измениться на D (но не наоборот). Обычно инструкция получает S-разметку, если все вложенные инструкции и подвыражения также имеют S-разметку. Исключением из этого правила является инструкция `return`, которая всегда имеет D-разметку.

В случае, если инструкция `if-then-else` получает D-разметку условия, то все присваивания в теле этой инструкции (в ветках `then` и `else`) получают D-разметку. При этом для переменных и полей примитивного типа, находящихся слева от знака присваивания, присваивание производится так, как будто правая часть имеет D-разметку (см. подраздел 2.2). Для переменных и полей ссылочного типа, находящихся слева от знака присваивания, присваивание производится так, как будто правая часть имеет в качестве разметки D-BT-объект (см. подраздел 2.3). Данное правило необходимо, поскольку если `if-then-else` имеет D-условие, на этапе анализа неизвестно, истинно ли условие или ложно, а значит неизвестно, исполнятся ли присваивания в теле `if-then-else` или нет. Поэтому их необходимо перенести в остаточную программу. Инструкция `switch` анализируется аналогично `if-then-else`.

После окончания анализа инструкции управления, имеющей S-разметку, нужно выполнить процедуру слияния состояний. Подробнее о ней в разделе 4.

Отметим, что инструкции `for`, `while` и `do-while`, т. е. циклы, анализируются несколько раз, до тех пор, пока разметка цикла на N и N+1 итерации на окажется одинаковой. Каждая итерация разметки тела цикла аналогична анализу инструкции `if-then-else`.

Существенным отличием алгоритма анализа времени связывания, рассматриваемого в этой статье (алгоритм JaSpe), от алгоритма лежащего в основе CILPE является то, что алгоритм, реализованный в JaSpe, анализирует дерево абстрактного синтаксиса (abstract syntax tree, AST) программы, а CILPE работает с графом потока управления (control flow graph). Важно также то, что JaSpe разработан так, чтобы результат был понятен пользователю, а специализатор CILPE генерирует код, не предназначенный для чтения пользователем. В результате, специализатор CILPE на этапе BTA может проводить ряд оптимизаций (например, некоторое подобие раскрытия циклов, приводящее к росту программы), в то время как JaSpe придерживается более консервативного подхода, когда на этапе BTA производится только анализ, без каких-либо оптимизаций, чтобы результат BTA был понятен пользователю.

Кроме того, в языке SOOL (Stack-based Object-Oriented Language) [6], на котором описаны программы, анализируемые CILPE, отсутствуют явные инструкции-циклы. Чтобы организовать цикл в SOOL, нужно использовать инструкции условного и безусловного перехода.

В связи с двумя рассмотренными выше отличиями анализ циклов в JaSpe существенно переработан и модифицирован по сравнению с CILPE.

BT-значение инструкций в процессе разметки программы, которая проводится рассматриваемым в данной статье алгоритмом, может только увеличивается в соответствии с отношением порядка в решетке {undefined, S, D}, где undefined $<$ S $<$ D.

2.1. Анализ выражений

При вычислении BT-значения многих инструкций языка Java необходимо проводить анализ подвыражений. Подвыражения (как и выражения) в контексте BTA можно разделить на две группы: (под)выражения примитивного типа и (под)выражения ссылочного типа. В Java восемь примитивных типов: `char`, `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`.

Перечислим те выражения, которые в рассматриваемом подмножестве языка Java могут иметь ссылочный тип: обращение к

имени переменной, присваивание в переменную, обращение к полю, присваивание в поле, обращение к элементу массива, присваивание в элемент массива и выражение создания объекта (оператор new).

2.2. Анализ выражений примитивного типа

При анализе выражений примитивного типа используются следующие правила:

- константы получают S-разметку;
- все обращения к статическим примитивным полям классов получают D-разметку;
- при старте специализации аргументы метода, являющегося точкой входа в специализацию, получают D-разметку;
- выражение примитивного типа, аналогично инструкции языка, изначально получает S-разметку. На основе BT-значений подвыражений строиться новое BT-значение. Исключение составляют обращение к полю объекта, которое имеет примитивный тип. Этот случай рассматривается в разделе 2.3, посвященном анализу выражений ссылочного типа, поскольку обращение к полю подразумевает анализ BT-объекта, к полю которого осуществляется доступ;
- если объявление переменной не имеет правой части, то оно получает S-разметку. При этом в таблицу BT-значений переменных (BT-окружение) записывается пара, состоящая из имени переменной и примитивного S-значения;
- объявление переменной или присваивание с правыми частями, получившими S-разметку, также получают S-разметку. Иначе они получают D-разметку. В данных обоих случаях в BT-окружение записывается пара, состоящая из имени переменной и BT-разметки правой части;
- присваивание в аргумент метода и обращение к значению аргумента метода обрабатываются аналогично присваиванию в переменную и обращению к значению переменной;
- обращения к значению переменной получают ту BT-разметку, которая хранится в BT-окружении и соответствует имени данной переменной;
- унарные и бинарные операции получают S-разметку, если все их подвыражения-аргументы имеют S-разметку. В остальных случаях унарные и бинарные операции получают D-разметку. Отметим, что анализ унарных и бинарных операций в специализаторе JaSpe отличается от анализа того же типа операций в

CILPE: в последнем подразумевается, что данные операции выполняются над значениями в стеке, а не над переменными или полями объектов, как в JaSpe.

BT-разметка каждого из выражений примитивного типа в процессе анализа времени связывания может только увеличивается в соответствии с решеткой {undefined, S, D}, где undefined $<$ S $<$ D.

2.3. Анализ конструкций ссылочного типа

В контексте BTA конструкции ссылочного типа можно разделить на две группы. Первая группа связана с созданием BT-объектов, а вторая — с получением доступа к уже созданным BT-объектам. Новые BT-объекты создаются в следующих основных случаях:

- при анализе вхождения e выражения вида `new Clazz()`, если в списке созданных BT-объектов ещё нет BT-объекта, соответствующего данному выражению, то создаётся новый S-BT-объект b (причем все примитивные поля данного объекта также инициализируются S-разметкой, а ссылочные остаются не инициализированными) и он возвращается в качестве результата разметки этого выражения. При этом в список созданных BT-объектов помещается отображение

  $$e \mapsto b,$$

  В случае специализатора JaSpe в качестве вхождения e используется объект (экземпляр класса), обозначающий узел в AST-дереве программы.
  В данном варианте алгоритма анализа подразумевается, что класс `Clazz` не переопределяет умолчательный конструктор, а выражение вида `new Clazz(...)` никогда не содержит аргументов внутри скобок. Также стоит упомянуть, что анализу подвергаются только те классы, чей исходный код доступен специализатору. Если исходный код класса не доступен специализатору, то все конструкции имеющие этот ссылочный тип размечаются D-BT-объектами;
- в момент начала анализа метода, являющегося точкой входа, для аргументов точки входа создаются D-BT-объекты;
- при старте BTA для статических полей ссылочного типа создаются D-BT-объекты.

Анализ ссылочных выражений, осуществляющих доступ к уже созданным BT-объектам, выполняется по следующим правилам:

- при анализе выражения `new Clazz()`, если в списке созданных BT-объектов есть BT-объект, соответствующий данному выражению, то в качестве разметки возвращается этот BT-объект;
- если объявление переменной не имеет правой части, то оно получает S-BT-объект в качестве разметки. При этом в таблицу BT-значений переменных (*BT-окружение*) записывается пара, состоящая из имени переменной и упомянутого S-BT-объекта;
- присваивания и объявления переменных, имеющие правую часть, получают ту же разметку, которую имеет правая часть. В BT-окружение записывается пара, состоящая из имени переменной и разметки правой части. Отметим, что поскольку речь идет о выражениях ссылочного типа, правая часть должна иметь разметку в виде BT-объекта;
- выражения обращения к значению переменной получают ту BT-разметку, которая хранится в BT-окружении и соответствует данной переменной;
- присваивание в аргумент метода и обращение к значению аргумента метода обрабатываются аналогично присваиванию в переменную и обращению к значению переменной;
- рассмотрим обращение к полю (имеющему вид `object.field`). Для выражения `object` по описанной процедуре построено BT-значение. Здесь `object` — либо имя переменной, либо имя аргумента метода, либо обращение к элементу массива, либо (рекурсивно) обращение к полю вида `object.field`, рассмотрению которого посвящен данный пункт. Выражение `object` имеет ссылочный тип и для ссылочного типа BT-значение является BT-объектом, включающим в себя пары, состоящие из квалифицированного имени поля и его разметки. Чтобы получить разметку обращения к полю, в этом списке пар производится поиск по квалифицированному имени поля, вычисленному еще до начала BTA для имени field при построении AST. Найденное в списке значение и будет BT-значением обращения к полю;
- рассмотрим присваивание в поле объекта (которое имееет вид `object.field = value`). Аналогично обращению к полю, ищется BT-объект для `object`, а затем BT-значение поля `object.field`. После этого производится слияние BT-значений поля и правой части присваивания. Если поле, в которое выполняется присваивание, имеет примитивный тип и оно изменило своё значение с S на D, то анализ возвращается («откатывается») на выражение, в котором был создан BT-объект, найденный для выражения object. Если поле, в которое выполняется присваивание, имеет

ссылочный тип, то в результате слияния также может произойти возврат анализа времени связывания на выражение, в котором был создан один из двух рассматриваемых объектов (для левой и правой части присваивания). Подробнее о слиянии и об откате можно прочитать в разделе 4;

- выражение вида `obj instanceof Clazz` получает S-разметку, если для выражение `obj` было аннотировано S-BT-объектом, иначе выражение `obj instanceof Clazz` получает D-разметку;
- при анализе обращения к элементу массива учитывается тот факт, что всегда используется одна и та же разметка для всех элементов одного массива. Поэтому индекс элемента игнорируется и анализ обращения к элементу массива аналогичен анализу обращения к переменной;
- присваивание в любой из элементов массива рассматривается как присваивание в переменную, имеющую такую же разметку, как у массива (массив имеет одну разметку для всех его элементов).

3. Моновариантность BT-объектов

В рассматриваемом алгоритме анализа времени связывания BT-объекты моновариантны. Это означает, что после завершения BTA любой BT-объект всегда имеет единственную разметку во всех точках/местах программы.

В процессе BTA некоторые поля BT-объекта могут менять разметку с S на D (но не наоборот). В этом случае происходит откат алгоритма на инструкцию, в которой был создан BT-объект, чья разметка изменилась. После чего начиная с этой инструкции происходит переразметка программы (т. е. разметка выполняется заново). Переразметка необходима, поскольку изменение значений, хранящихся в списке полей каждого из BT-объектов, глобально — разметка должна «обновиться» и при этом быть одинаковой во всех точках программы, где данный BT-объект используется.

Нужно отметить, что в процессе переразметки при обработке выражения вида `new Clazz()` алгоритм BTA извлекает разметку «старых» BT-объектов из списка созданных BT-объектов (а не создаёт новый S-BT-объект), если на предыдущих итерациях разметки для этого выражения уже был создан некоторый BT-объект. Этот факт ограничивает число возможных переразметок: при анализе программы существует ограниченное число BT-объектов, подверженных переразметке (по количеству выражений `new`).

Утверждение 1. *Количество переразметок ограничено.*

Обоснование. Если не учитывать момент старта работы алгоритма BTA, BT-объекты порождаются только в ходе выполнения анализа выражений вида `new Clazz()`. Без учёта переразметки каждая из инструкций анализируется однократно. При переразметке выражения вида `new Clazz()` оно размечается тем BT-объектом, который был построен на одной из предыдущих итераций и хранится в списке созданных BT-объектов. Поэтому общее число BT-объектов не превышает количество выражений вида `new Clazz()` в программе. При этом у каждого BT-объекта ограниченное число полей. Переразметка происходит при переходе разметки поля из S в D, а переход разметки поля из D в S невозможен. В итоге: количество BT-объектов ограничено, количество полей у них тоже ограничено, переразметка может возникать только один раз для каждого поля, поэтому количество переразметок ограничено. □

```
1 C obj = new C();
2 obj.x = 1;            //поле obj.x - D, хотя правая часть S
3 int y = obj.x;        // y имеет также D-разметку, т.к. x - D
4 obj.x = arg;          // аргумент метода имеет D-разметку
```

Рисунок 1. Моновариантность BT-объектов

Рассмотрим пример моновариантной разметки BT-объектов на примере (см. рисунок 1).

На рисунке 1 жирным шрифтом выделены сущности, имеющие S-разметку, а подчеркиванием — имеющие D-разметку.

В строке 2 поле `obj.x` имеет D-разметку, несмотря на то, что правая часть присваивания имеет S-разметку (константа «1» размечается как S). Причиной этого является моновариантность BT-объектов и тот факт, что в строке 4 разметка поля `obj.x` становится D. Как следствие происходит откат на выражение создания BT-объекта, соответствующего переменной `obj`, которое находится в строке 1 (оператор `new C()`). После этого на протяжении выполнения алгоритма анализа всей программы поле `obj.x` будет иметь D-разметку.

4. Слияние BT-значений

При анализе присваивания в поле, а также после анализа инструкций управления происходит слияние BT-значений. При слиянии двух BT-значений x и y находится их верхняя грань z.

Слияние после выполнения анализа инструкций управления производится только над разметками, описывающими переменные (но не поля). При разметке полей слияние происходит в момент анализа присваиваний.

Верхняя грань для BT-значений примитивного типа находится достаточно просто: если одно из BT-значений x или y имеет разметку D, то результирующая разметка будет D, иначе результирующая разметка S.

При слиянии двух BT-объектов b_0 и b_1, находится верхняя грань c двух разметок по следующим правилам:

(*1*) В начале слияния b_0 и b_1, происходит запоминание того факта, что эти BT-объекты уже подвергаются слиянию друг с другом. Это необходимо, поскольку при слиянии полей BT-объектов b_0 и b_1, может получиться, что данные BT-объекты будут рассмотрены как кандидаты на слияние ещё раз. В последнем случае, повторного слияния b_0 и b_1 не произойдёт, поскольку анализ помнит, что эти два BT-объекта уже подвергаются слиянию.

(*2*) В BT-объект c добавляются все инициализированные поля BT-объекта b_0.

(*3*) Происходит добавление инициализированных полей BT-объекта b_1 в BT-объект c. Если квалифицированное имя поля из списка BT-объекта b_1 уже встречается в BT-объекте c, происходит слияние этих двух полей.

(*4*) Во второй компонент тройки BT-объекта c, представляющий из себя список классов, чью разметку описывает данный BT-объект, добавляются все классы из вторых компонентов BT-объектов b_0 и b_1. В случае, если один и тот же класс встречается в определениях b_0 и b_1, то он добавляется только в одном экземпляре.

(*5*) Разметка верхнего уровня, которая является первым компонентом тройки в определении BT-объекта, определяется так: если у одно из BT-объектов b_0 или b_1 разметка верхнего уровня равна D, то разметка верхнего уровня BT-объекта c равна D, иначе она равна S.

После того, как найдена верхняя грань BT-объектов b_0 и b_1, они заменяются на c (в том числе в списке созданных BT-объектов и в BT-куче). В случае, если при слиянии какое-либо поле изменило свою разметку с S на D, происходит возврат анализа на момент создания того BT-объекта, чьё поле изменило разметку. Если таких BT-объектов

несколько, возврат анализа происходит на самый ранний из моментов создания.

Далее рассмотрим примеры слияния BT-объектов.

4.1. Пример слияния BT-объектов: присваивания

```
1 C c = new C();
2 C obj0 = new C();
3 c.next = obj0;
4 obj0.x = 1;        // поле x имеет D-разметку
5 int y = c.next.x; // y имеет D-разметку
6 C obj1 = arg;      // в obj1 присваивается объект с D-разметкой
7 c.next = obj1;     // слияние obj0 с obj1
```

Рисунок 2. Слияние BT-объектов при присваивании

На рисунке 2 жирным шрифтом выделены сущности, имеющие S-разметку, а подчеркиванием — D-разметку. Комментарии относятся к процессу анализа времени связывания.

Пусть дан класс C, который имеет два поля: поле `next`, указывающее на следующий элемент списка, и поле `x` — целое число. Рассмотрим пример с таким классом C, который иллюстрирует эффект от слияния (рисунок 2).

Назовём b_c, b_{obj0} и b_{obj1} BT-объекты, описывающие разметку объектов, на которые ссылаются переменные `c`, `obj0` и `obj1` соответственно. В строке 4 происходит присваивание в поле `obj0.x` константы, имеющей S-разметку, однако поле `obj0.x` всё равно имеет D-разметку. Этот факт можно пояснить следующим образом: во-первых, происходит слияние BT-объектов b_{obj0} и b_{obj1} в процессе анализа строки 7; во-вторых, BT-объект b_{obj1}, имеет D-разметку поля x, как и весь BT-объект (строка 6).

Слияние происходит из-за того, что BT-объект b_c моновариантен, как и все BT-объекты. Это означает, что данный BT-объект имеет одну и ту же разметку на протяжении всей программы. Это, в свою очередь означает, что разметка BT-объекта b_c одна и та же для строк 3 и 7, следовательно, и разметка поля `c.next` для строк 3 и 7 одна и та же. Последний факт влечёт за собой то, что объекты, на которые ссылаются переменные `obj0` и `obj1`, должны иметь одинаковую разметку (ту же самую, что и `c.next`). Поэтому для них строиться один BT-объект, который и является разметкой, а тот факт что он один для обоих объектов, означает, что разметка одинаковая.

4.2. Пример слияния BT-объектов: инструкции управления

Другим случаем, в котором происходит слияние BT-объектов, является анализ инструкции `if-then-else`. Пусть производится анализ инструкции `if-then-else`, размеченной как S. Пусть также после окончания анализа ветки `then`, оказывается, что разметка некоторой переменной описывается BT-объектом b_{obj0}, А после анализа ветки `else`, разметка этой же переменной описывает BT-объектом b_{obj1}.

BT-объекты b_{obj0} и b_{obj1} будут слиты в новый BT-объект c после окончания анализа инструкции `if-then-else`.

Такое слияние выполняется, поскольку специализатор на этапе BTA не может выяснить, какая из разметок b_{obj0} или b_{obj1} будет актуальна на этапе генерации остаточной программы, поскольку при BTA не известно конкретное значение условия (истинно оно или ложно), а только его разметка — S. Значит, на этапе анализа времени связывания отсутствует информация о том, какая из двух веток — `then` или `else` — будет выполнена, а значит неизвестно, какая из двух разметок b_{obj0} или b_{obj1} будет у ссылочной переменной. Поэтому эти две разметки нужно объединить в одну, по описанным выше правилам.

Для остальных инструкций управления слияние производится аналогично инструкции `if-then-else` (выполняется слияние состояний переменных на момент старта анализа инструкции и момент окончания итерации анализа инструкции).

```
1 C obj0;
2 if (s) {            // переменная s имеет S-разметку
3   obj0 = new C();
4   obj0.x = 1;
5 } else {
6   obj0 = new C();
7   obj0.x = arg;     // arg - аргумент, имеет D-разметку
8 }
9 int y = obj0.x;
```

Рисунок 3. Слияние BT-объектов при анализе if-then-else

Рисунок 3 иллюстрирует слияние для инструкции `if-then-else`. В строках 3 и 6 создаются новые объекты, для каждого из которых при выполнении алгоритма BTA строится свой BT-объект. В строке 4 поле `obj0.x` имеет D-разметку, хотя в него присваивается константа, имеющая S-разметку, поскольку после анализа инструкции `if-then-else` в строке 8 происходит слияние BT-объектов, созданных при анализе строк 3 и 6 (назовём эти объекты b_{obj0} и b_{obj1} соответственно). BT-объект b_{obj1} получает D-разметку поля `obj0.x` (строка 7). В результате верхняя

грань для b_{obj0} и b_{obj1} имеет D-разметку поля `obj0.x`. После чего оба этих BT-объекта заменяются на верхнюю грань и происходит переразметка программы начиная со строки 3. В итоге поле `obj0.x` имеет разметку D на протяжении всей программы.

5. Обзор области и сравнение

Специализатор MIX [12] был первым (из известных нам) специализатором, выделившим анализ времени связывания в отдельный проход, размечающий программные конструкции как статические/динамические. MIX предназначен для специализации программ, реализованных на чисто функциональном языке Mixwell. Этот язык похож на языки Scheme и Lisp. Одним из главных требований, которые рассматривались при реализации специализатора MIX, была самоприменимость. Самоприменимость была необходима для исследования возможности порождения эффективных генераторов компиляторов. В работе [MIX] исследуется этот вопрос. Однако, стоит отметить, что генераторы компиляторов получались громоздкими, а логика их работы оказывалась трудной для понимания человеком.

Принципы, изложенные в [12], получили своё развитие в специализаторе Unmix [13]. В специализаторе Unmix были предложены и реализованы некоторые усовершенствования по сравнению с MIX, в частности — использование разных представлений для S- и D-значений при генерации остаточной программы, а также - автоматический повышатель арности/местности. В результате, размер генератора компиляторов уменьшился на порядок, а его структура стала очевидной.

Изложенный выше алгоритм анализа времени связывания опирается на принцип, введенный в работе [12], — разделение программных конструкций на статические и динамические. Как и в специализаторе MIX, алгоритм BTA в настоящей работе основан на абстрактной интерпретации. Также, мы использовали тот же способ описания алгоритма BTA, как в работе [12]: приводятся правила разметки конструкций, основанные на абстрактной интерпретации. Однако, специализатор MIX работает с функциональным языком, поэтому остальная часть принципов, изложенных в [12], в настоящей работе не использовалась.

Методы специализации развивались и для императивных языков, в частности, для языка C — одного из наиболее популярных языков данного класса. Для этого языка разработано два основных специализатора: C-MIX и Tempo.

C-MIX [14, 15] был первым (насколько нам известно) специализатором языка C и был разработан в начале 1990-х годов. Анализу времени связывания в C-MIX предшествуют анализ указателей (pointer analysis) и анализ побочных эффектов (side-effect analysis). Цель анализа указателей — для каждой переменной-указателя найти области памяти, на которые данная переменная может ссылаться. Если в процессе анализа обнаруживается, что несколько указателей могут ссылаться на одну и ту же область памяти, они аннотируются одинаково.

Другой этап анализа, предваряющий BTA в C-MIX — это анализ побочных эффектов. Анализ побочных эффектов обнаруживает инструкции и выражения, которые могут иметь побочные эффекты на этапе исполнения программы. Как утверждают авторы C-MIX, при статическом анализе невозможно точно определить, имеет ли конструкция побочный эффект. C-MIX вычисляет приближение: некоторые конструкции, не имеющие побочного эффекта на этапе исполнения программы, могут быть отмечены, как имеющие побочный эффект.

Анализ времени связывания в C-MIX строится на основе решения системы ограничений (constraint solving). Этот вариант BTA состоит из четырёх этапов. На первом этапе специализатор обходит программу и строит систему ограничений. На втором этапе производится нормализация этой системы. На третьем этапе она решается, а на четвёртом программа размечается в соответствии с полученным решением.

В работе [16] рассматривается аспекты реализации анализа времени связывания для подмножества императивного языка C. Эта реализация входит в состав специализатора Tempo. BTA в специализаторе Tempo требует предварительного проведения анализа синонимов (alias analysis) и анализа определений (definition analysis). Необходимость проведения дополнительных этапов анализа связана с наличием в языке C указателей. Анализ синонимов в Tempo выполняет ту же задачу, что и анализ указателей в C-MIX. Анализ определений находит для каждой инструкции в программе множество областей памяти, в которые выполняются присваивания внутри данной инструкции.

В книге [17] специализация рассматривается с позиции разработчика Tempo. В ней утверждается, что в Tempo используется подход к BTA, основанный на анализе потока данных (data flow analysis).

Анализ времени связывания, предлагаемый в нашей работе, не требует никаких дополнительных этапов, в отличие от BTA в Tempo или C-MIX. Анализ в JaSpe основан на абстрактной интерпретации, а

не на анализе потока данных (как в Tempo) или решении системы ограничений (как в C-MIX).

Наиболее близкими к обсуждаемом в данной статье алгоритму являются работы У.П. Шульца и Ю.А. Климова, посвящённые специализации программ на объектно-ориентированных языках.

У.П. Шульц в рамках своей работы над специализацией объектно-ориентированных языков [18–20] разработал алгоритм анализа времени связывания для подмножества языка Java. Это подмножество включает описание классов, конструкторов и методов, операции над примитивными данными и виртуальные вызовы. Однако, это подмножество имеет ряд существенных ограничений: во-первых, отсутствует возможность менять значение полей объектов. Поле получает своё значение только в конструкторе, присваивание в поле внутри метода отсутствует. Во-вторых, тело метода в исследуемом Шульцем подмножестве языка Java состоит из одного выражения и не может содержать последовательность инструкций.

Алгоритм анализа времени связывания в работе Шульца представлен в виде правил построения системы ограничений на разметку, которая должна быть решена при помощи известных методов. Однако, весьма существенные ограничения на подмножество Java ставят под сомнение возможность использования предложенного Шульцем алгоритма BTA для реальных задач.

Алгоритм анализа времени связывания в данной статье определён на ином подмножестве Java, чем алгоритм BTA приведённый Шульцем. Алгоритм, которому посвящена данная статья, поддерживает изменяемые объекты, в отличии от алгоритма, разработанного Шульцем. Алгоритм Шульца может обрабатывать вызовы методов. В будущем мы рассчитываем, что и наш алгоритм также сможет анализировать вызовы методов, для этого созданы все предпосылки (требуется адаптировать алгоритм, разработанный, Климовым, см. ниже). Также отметим, что алгоритм BTA у Шульца основывается на построении и решении системы ограничений на разметку, в то время как алгоритм, которому посвящена данная статья, основан на абстрактной интерпретации.

Некоторые результаты исследования, которое доводит частичные вычисления до готовности к практическому применению, изложены в работах Ю.А. Климова [4–11]. Климов в рамках своей диссертационной работы разработал специализатор CILPE для языка SOOL (Stack-based Object-Oriented Language). Язык SOOL [6] является подмножеством языка CIL [21], а язык CIL можно назвать «байткодом» платформы Microsoft .NET.

Алгоритм BTA, изложенный в данной статье, является адаптацией алгоритма анализа времени связывания реализованного в специализаторе CILPE. В своей диссертации Климов вводит понятие BT-кучи и понятие разметки программы, опирающееся на понятие BT-кучи; данные понятия мы заимствуем в своём исследовании. Конструкции, содержащиеся в языке SOOL, аналогичны базовым конструкциям объектно-ориентированных языков Java и C#. Однако, многие конструкции в SOOL имеют специфику — они рассчитаны на работу со стеком, а не с переменными или кучей. Также в языке SOOL нет циклов в том виде, в котором они присутствуют в языке Java: в SOOL циклы организуются на основе инструкций условного и безусловного перехода. В связи с этими двумя особенностями принципы и алгоритмы, разработанные Климовым, требуют адаптации для применения их к языку Java.

В своих публикациях Климов уделяет большое внимание корректности разметки и не приводит конкретного алгоритма BTA. Данная работа призвана улучшить сложившуюся ситуацию и продолжает [4, 5] в направлении разработки принципов и алгоритмов в части специализации программ на широко распространенном объектно-ориентированном языке Java.

Современной разработкой, опирающейся на метод частичных вычислений, является фреймворк Truffle, входящий в набор инструментов GraalVM [22, 23] . Данный инструмент предназначен для реализации интерпретаторов для высокопроизводительных динамических или предметно-ориентированных языков. Высокая производительность достигается за счет использования частичных вычислений: интерпретатор реализуемого языка специализируется по части программы, в результате чего получается откомпилированная версия этой части программы, в соответствии с первой проекцией Футамуры [1]. Полученный код исполняется более эффективно, чем исходный в интерпретаторе. Основная задача частичных вычислений в GraalVM — предоставить разработчику интерпретатора легкий путь получения технологии, аналогичной JIT-компиляции, с минимальным приложением усилий со стороны разработчика. Частичные вычисления в GraalVM принадлежат к так называемому «онлайновому» виду, т. е. не содержат этапа анализа времени связывания. Реализованный в GraalVM подход привёл к тому, что, на данный момент, в общем случае, нельзя гарантировать терминируемость процесса специализации в GraalVM. Поэтому проблема обеспечения терминируемости перекладывается на

плечи программиста-пользователя этой системы — он должен описать интерпретатор так, чтобы специализация не зацикливалась.

Заключение

В статье был рассмотрен поливариантный алгоритм анализа времени связывания для программ на подмножестве объектно-ориентированного языка Java версии 8. Данный алгоритм предназначен для внутрипроцедурного анализа и основывается на абстрактной интерпретации. В отличие от предшествующих аналогов, работающих с объектно-ориентированными языками, представленная в данной статье версия алгоритма обрабатывает более широкое множество программных конструкций на основе информации, доступной в границах одного Java-метода. Приводятся определения BT-объекта, BT-кучи и разметки программы.

Представленный алгоритм анализа времени связывания является:

- поливариантным по переменным;
- поливариантным по классам;
- моновариантным по операциям.

В будущем авторы планируют:

- реализовать межпроцедурную оптимизацию;
- расширить возможности анализа времени связывания для применения к обработке конструкций языка, работающих с исключениями;
- адаптировать алгоритм для более современных версий языка Java.

Исполняемая программа. Представленный в данной статье алгоритм BTA доступен в виде набора плагинов для Eclipse IDE по адресу: `ftp://ftp.botik.ru/rented/iaadamovich/JaSpe_intraP_BTA/`.

Благодарности

Авторы выражают благодарность Сергею Романенко, Аркадию и Андрею Климовым за ценные советы по методу частичных вычислений и конструктивную критику принятых решений при реализации специализатора JaSpe.

Список литературы

[1] Y. Futamura. “Partial evaluation of computation process — an approach to a compiler-compiler”, *Higher-Order and Symbolic Computation*, **12**:4 (1999), pp. 381-391. ↑4,22

[2] N.D. Jones, C.K. Gomard, P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-hall International Series in Computer Science, Prentice-Hall, 1993, ISBN 978-0130202499, 415 pp. ↑4

[3] I.A. Adamovich, And.V. Klimov. «Интерактивный специализатор подмножества языка Java, основанный на методе частичных вычислений», *Proceedings of the Institute for System Programming*, **30**:4 (2018), с. 29-44 (in English). ↑5

[4] Ю.А. Климов. *Специализация программ на объектно-ориентированных языках методом частичных вычислений*, дис. к.ф.-м.н., Институт прикладной математики им. М.В. Келдыша РАН, М., 2009. ↑6,7,21,22

[5] Ю.А. Климов. «Специализатор CILPE: анализ времен связывания», *Препринты ИПМ им. М. В. Келдыша*, 2009, 007, 28 с. ↑6,7,21,22

[6] Ю. А. Климов. «SOOL: объектно-ориентированный стековый язык для формального описания и реализации методов специализации программ», *Препринты ИПМ им. М. В. Келдыша*, 2008, 044, 32 с. ↑6,7,10,21

[7] Ю.А. Климов. «Особенности применения метода частичных вычислений к специализации программ на объектно-ориентированных языках», *Препринты ИПМ им. М. В. Келдыша*, 2008, 012, 27 с. ↑6,7,21

[8] Ю.А. Климов. «Возможности специализатора CILPE и примеры его применения к программам на объектно-ориентированных языках», *Препринты ИПМ им. М. В. Келдыша*, 2008, 030, 28 с. ↑6,7,21

[9] Ю.А. Климов. «Специализатор CILPE: генерация остаточной программы», *Препринты ИПМ им. М. В. Келдыша*, 2009, 008, 26 с. ↑6,7,21

[10] Ю.А. Климов. «Специализатор CILPE: доказательство корректности», *Препринты ИПМ им. М. В. Келдыша*, 2009, 033, 32 с. ↑6,7,21

[11] Ю.А. Климов. «Специализатор CILPE: частичные вычисления для объектно-ориентированных языков», *Программные системы: теория и приложения*, **1**:3(3) (2010), с. 13-36. ↑6,7,21

[12] 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. ↑19

[13] С.А. Романенко. «Генератор компиляторов, порожденный самоприменением специализатора, может иметь ясную и естественную структуру», *Препринты ИПМ им. М. В. Келдыша*, 1987, 026, 39 с. ↑19

[14] L.O. Andersen. *Program analysis and specialization for the C programming language*, Ph.D. dissertation, DIKU, University of Copenhagen, 1994. ↑20

[15] L.O. Andersen. “Binding-time analysis and the taming of C pointers”, *Proceedings of the 1993 ACM SIGPLAN symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM’93, ACM, 1993, pp. 47-58. ↑20

[16] L. Hornof, J. Noye. “Accurate binding-time analysis for imperative languages: flow, context, and return sensitivity”, *Theoretical Computer Science*, **248**:1–2 (2000), pp. 3-27. ↑20

[17] R. Marlet. *Program Specialization*, Wiley-ISTE, 2012, ISBN 9781848213999, 544 pp. ↑20

[18] U.P. Schultz, J.L. Lawall, C. Consel. “Automatic program specialization for Java”, *ACM Trans. Program. Lang. Syst.*, **25**:4 (2003), pp. 452-499. ↑21

[19] U.P. Schultz. “Partial evaluation for class-based object-oriented languages”, PADO 2001: Programs as Data Objects, Lecture Notes in Computer Science, vol. **2053**, Springer, 2001, ISBN 978-3-540-42068-2, pp. 173-197. ↑21

[20] U.P. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation*, Ph.D. dissertation, University of Rennes I, Rennes, France, 2000. ↑21

[21] А.В. Макаров, С.Ю. Скоробогатов, А.М. Чеповский. *Common Intermediate Language и системное программирование в Microsoft.NET*, Интернет-Университет Информационных Технологий (ИНТУИТ), Бином. Лаборатория знаний, М., 2006, ISBN 978-5-94774-735-5, 328 с. ↑21

[22] 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*, ACM, New York, NY, USA, 2013, pp. 187-204. ↑22

[23] 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”, *SIGPLAN Not.*, **52**:6 (June 2017), pp. 662-676. ↑22

Поступила в редакцию 17.01.2020
Переработана 11.02.2020
Опубликована 20.02.2020

Рекомендовал к публикации *д.ф.-м.н. С. М. Абрамов*

Пример ссылки на эту публикацию:

И. А. Адамович, Ю. А. Климов. «Специализатор JaSpe: алгоритм внутрипроцедурного анализа времени связывания программ на подмножестве языка Java». *Программные системы: теория и приложения*, 2020, **11**:1(44), с. 3–29. 10.25209/2079-3316-2020-11-1-3-29

http://psta.psiras.ru/read/psta2020_1_3-29.pdf

Об авторах:

Игорь Алексеевич Адамович

Младший научный сотрудник ИПС им. А. К. Айламазяна РАН. Принимал активное участие в разработке коммуникационных сетей «Паутина» и «3D-тор» суперкомпьютера «СКИФ-Аврора».

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 для программ на промежуточном объектно-ориентированном языке CIL платформы MS.NET, принимал активное участие в разработке коммуникационных сетей «Паутина» и «3D-тор» суперкомпьютера «СКИФ-Аврора» и программного обеспечения для сетей «МВС-Экспресс» и SCI.

0000-0001-5081-1547
e-mail: yuklimov@keldysh.ru

CSCSTI 50.05.09, 50.41.17
UDC 519.681.3

Igor A. Adamovich, Yuri A. Klimov. *The JaSpe specializer: an algorithm of intra-procedural binding time analysis for programs in Java language subset.*

Abstract. A binding-time analysis in partial evaluation aimed at optimizing programs divides software constructs into static and dynamic. A specializer executes static constructs, and transfer dynamic ones into the resulting code. Currently, partial evaluation is mainly used for the non-trivial compilation of programs without a compiler, with only an interpreter and a specializer. As previous studies have shown, the effectiveness of such an application of the partial evaluation method significantly depends on the quality of the program annotation obtained by performing the binding-time analysis.

The paper is devoted to the features of the binding-time analysis algorithm. The features that arose during algorithm implementation for the widespread object-oriented Java language within the JaSpe specializer developed by the authors of this publication. The paper describes the basic concepts of the binding-time analysis implemented and the intra-procedural version of the algorithm. The article also discusses the algorithm details related to the program constructs that use reference data types.

Apart from previous counterparts for object-oriented languages, this algorithm non-trivially handles some language constructs: branches (if, switch), loops (for, while, do), and block instructions that contain a sequence of other instructions. Unlike the similar algorithms that work with imperative and functional languages, the considered algorithm uses the BT-objects, which allow specializer to obtain more accurate annotation (with a higher fraction of static constructs) when processing object-oriented programs. Another feature of this algorithm is the focus on interactivity and readability of results.

Key words and phrases: modern programming languages, static program analysis, program transformation, metaprogramming, mixed computation, interactive specialization.

2010 *Mathematics Subject Classification*: 97P30; 97P20, 97P40

References

[1] Y. Futamura. "Partial evaluation of computation process — an approach to a compiler-compiler", *Higher-Order and Symbolic Computation*, **12**:4 (1999), pp. 381-391. doi↑4,22

[2] N.D. Jones, C.K. Gomard, P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-hall International Series in Computer Science, Prentice-Hall, 1993, ISBN 9780130202499, 415 pp. URL↑4

[3] I.A. Adamovich, And.V. Klimov. "An interactive specializer based on partial evaluation for a Java subset", *Proceedings of the Institute for System Programming*, **30**:4 (2018), pp. 29-44. ↑5

[4] Yu.A. Klimov. *Program specialization for object-oriented languages by partial evaluation*, dis. k.f.-m.n., Institut prikladnoy matematiki im. M.V. Keldysha RAN, M., 2009 (in Russian). ↑6,7,21,22

[5] Yu.A. Klimov. "Specializer CILPE: binding time analysis", *Preprinty IPM im. M. V. Keldysha*, 2009, 007 (in Russian), 28 pp. ↑6,7,21,22

[6] Yu. A. Klimov. "SOOL: an object-oriented stacked-based language for specification and implementation of program specialization techniques", *Preprinty IPM im. M. V. Keldysha*, 2008, 044 (in Russian), 32 pp. ↑6,7,10,21

[7] Yu.A. Klimov. "Program specialization for object-oriented languages by partial evaluation: approaches and problems", *Preprinty IPM im. M. V. Keldysha*, 2008, 012 (in Russian), 27 pp. ↑6,7,21

[8] Yu.A. Klimov. "Specializer CILPE: examples of object-oriented program specialization", *Preprinty IPM im. M. V. Keldysha*, 2008, 030 (in Russian), 28 pp. ↑6,7,21

[9] Yu.A. Klimov. "Specializer CILPE: residual program generation", *Preprinty IPM im. M. V. Keldysha*, 2009, 008 (in Russian), 26 pp. ↑6,7,21

[10] Yu.A. Klimov. "Specializer CILPE: correctness proof", *Preprinty IPM im. M. V. Keldysha*, 2009, 033 (in Russian), 32 pp. ↑6,7,21

[11] 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). ↑6,7,21

[12] 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. ↑19

[13] 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. ↑19

[14] L.O. Andersen. *Program analysis and specialization for the C programming language*, Ph.D. dissertation, DIKU, University of Copenhagen, 1994. ↑20

[15] L.O. Andersen. "Binding-time analysis and the taming of C pointers", *Proceedings of the 1993 ACM SIGPLAN symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM'93, ACM, 1993, pp. 47-58. ↑20

[16] L. Hornof, J. Noye. "Accurate binding-time analysis for imperative languages: flow, context, and return sensitivity", *Theoretical Computer Science*, **248**:1–2 (2000), pp. 3-27. ↑20

[17] R. Marlet. *Program Specialization*, Wiley-ISTE, 2012, ISBN 9781848213999, 544 pp. ↑20

[18] U.P. Schultz, J.L. Lawall, C. Consel. "Automatic program specialization for Java", *ACM Trans. Program. Lang. Syst.*, **25**:4 (2003), pp. 452-499. ↑21

[19] U.P. Schultz. "Partial evaluation for class-based object-oriented languages", PADO 2001: Programs as Data Objects, Lecture Notes in Computer Science, vol. **2053**, Springer, 2001, ISBN 978-3-540-42068-2, pp. 173-197. ↑21

[20] U.P. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation*, Ph.D. dissertation, University of Rennes I, Rennes, France, 2000. ↑21

[21] A.V. Makarov, S.Yu. Skorobogatov, A.M. Chepovskiy. *Common Intermediate Language and system programming in Microsoft.NET*, Internet-Universitet Informatsionnykh Tekhnologiy (INTUIT), Binom. Laboratoriya znaniy, M., 2006, ISBN 978-5-94774-735-5 (in Russian), 328 pp. ↑21

[22] 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*, ACM, New York, NY, USA, 2013, pp. 187-204. ↑22

[23] 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", *SIGPLAN Not.*, **52**:6 (June 2017), pp. 662-676. ↑22

Sample citation of this publication:

Igor A. Adamovich, Yuri A. Klimov. "The JaSpe specializer: an algorithm of intra-procedural binding time analysis for programs in Java language subset". *Program Systems: Theory and Applications*, 2020, **11**:1(44), pp. 3–29. (*In Russian*).

10.25209/2079-3316-2020-11-1-3-29

http://psta.psiras.ru/read/psta2020_1_3-29.pdf