ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ т. 12, № 4(51), с. 3–32

ББК 32.973.2
ГРНТИ 50.05.13
УДК 519.681.3

И. А. Адамович

Специализатор JaSpe: BT-объекты и межпроцедурный аспект алгоритма анализа времен связывания

Аннотация. Статья посвящена частичным вычислениям, использующим offline-стратегию. Мощность этого метода решения задачи специализации программ во многом зависит от анализа времен связывания, который размечает программные конструкции как выполнимые либо невыполнимые на этапе специализации.

Анализ времен связывания может использовать несколько вариантов разметки полей класса, зависящих от их использования в программе. Увеличивая число потенциальных оптимизаций, такая поливариантность по классам позволяет эффективно специализировать большее число программ. Наибольший эффект достигается на объектно-ориентированных языках, предполагающих создание большого количества различающихся по назначению экземпляров класса.

Известные алгоритмы анализа времен связывания расширяются до поливариантности по классам и распространяют их на объектно-ориентированный язык общего назначения. Новые методы реализованы в виде набора плагинов для Eclipse IDE, составляющих специализатор JaSpe для программ на Java.

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

Введение

Задача специализации программ подразумевает, что дана некоторая программа и известна часть значений ее аргументов и переменных. По этим данным строится новая программа, которая зависит уже только от той части аргументов, которая была неизвестна на стадии специализации. Часто новая программа более оптимальна: исполняется

© И. А. Адамович, 2021
© Институт программных систем имени А. К. Айламазяна РАН, 2021
© Программные системы: теория и приложения (дизайн), 2021

10.25209/2079-3316-2021-12-4-3-32

быстрее и/или использует меньше памяти. Требуется отметить, что исходная и специализированная программы должны быть эквивалентны, пока соблюдаются ограничения на известные аргументы.

Специализация программы может быть проведена даже если неизвестны все аргументы программы. В таком случае источником оптимизации будет информация, содержащаяся в самой программе: значения констант и знание конкретных ссылочных типов.

Задача специализации может быть решена разными методами, например суперкомпиляцией [1–4] или частичными вычислениями [5].

Важно отметить, что развитие методов специализации приводит к появлению новой продуктивной методологии разработки программ: программист будет разрабатывать универсальную программу, которая затем специализируется под конкретные условия применения. При этом специализатор гарантирует эквивалентность исходной и новой версий программы. Такой подход может повысить как производительность труда (исходная программа разрабатывается один раз для нескольких применений), так и качество результата: специализатор гарантирует отсутсвие ошибок в силу эквивалентности исходной программы и результата.

Настоящая статья излагает межпроцедурные особенности метода частичных вычислений, реализованного в специализаторе JaSpe. Данный инструмент разрабатывается с несколькими целями. Во-первых, сделать специализацию более доступной для понимания и использования рядовым программистом. Поэтому JaSpe погружен в Eclipse [6], популярную среду интегрированной разработки, и он специализирует программы, написанные на широко распространенном объектно-ориентированном языке Java. Второй целью является развитие метода частичных вычислений до состояния, когда его можно будет активно и регулярно применять к практическим задачам.

Научный вклад данной работы следующий: впервые приводятся методы, основанные на BT-объектах, поливариантного по классам анализа времен связывания. В область рассмотрения попадают конструкции объектно-ориентированного языка аналогичные вызову функции в функциональных или императивных языках. Такими конструкциями являются вызов метода, создания экземпляра класса, явные вызовы конструкторов.

Дальнейшее изложение следует следующей структуре. В разделе 1 вводятся основные понятия частичных вычислений, в том числе основанных на BT-объектах. В разделе 2 описываются методы анализа

времен связывания, ориентированные на обработку некоторых конструкций языка Java аналогичных вызову функции в функциональных языках. В разделе 3 приведен пример специализации программы, содержащей вызовы Java-методов, создания экземпляров классов и явные вызовы конструкторов. Раздел 4 содержит сравнение с близкими работами.

1. Частичные вычисления

Отличительной чертой частичных вычислений является то, что в процессе анализа или преобразования программы конструкции разделяются на два типа — вычислимые в статике (т. е. без исполнения программы) и не вычислимые в статике (вычислимые только в runtime). Метод частичных вычислений характеризуется агрессивным характером распространения констант (constant propagation) и генерации специализированных версий функций на основе распространяемых констант. Чтобы более детально описать особенности данного метода, следует выделить две стратегии частичных вычислений [**5**, **7**] — online и offline.

Online-стратегия частичных вычислений предполагает, что анализ и преобразование программы выполняются одновременно, за один проход. Таким образом, решение о том оптимизировать ли некоторую конструкцию принимаются на основе неполной (локальной) информации о программе. Локальность информации обусловлена тем, что при online-стратегии вычислений специализатор не может использовать информацию, содержащуюся в оставшейся, не обработанной части программы. При этом такой специализатор оперирует конкретными значениями переменных, а также значением “неизвестно”.

Частичные вычисления, работающие по offline-стратегии, обычно выполняются в два этапа. На первом этапе, называемом *анализ времен связывания* (binding-time analysis, BT-анализ), конструкции анализируемой программы разделяются на два типа — *статические* и *динамические*. Динамические конструкции перейдут в результирующую (*остаточную*) программу, а статические конструкции будут исполнены на втором этапе специализации, который называется *генерация остаточной программы*. Собственно, на втором этапе и происходят частичные вычисления.

Чтобы избежать конфликта таких терминов, как статический и динамический применительно к анализу времен связывания, с

понятиями статический в языке Java и динамический в смысле времени исполнения, мы будем использовать для терминов частичных вычислений сокращения S и D. Например: S-конструкция (статическая конструкция), D-вызов метода (вызов метода, имеющий динамическую разметку) и т. п.

1.1. Анализ времен связывания

Анализ времен связывания (BT-анализ) строит разметку программных конструкции, разделяя их на два вида: S и D. В процессе BT-анализа специализатор оперирует с абстрактными значениями S, D и некоторыми производными от них — BT-объектами. Конкретные числовые значения на данном этапе не используются.

Возможные значение разметки любой программной конструкции на этапе BT-анализа образуют решетку:

$$BT\text{-}types = \langle \{ Undefined, S, D \}, \leq \rangle.$$

Причем, отношение $\leq$ — отношение полного порядка:

$$Undefined \leq S \leq D.$$

BT-анализ может обладать свойством моновариантности или поливариантности. Моновариантный анализ отличается тем, что каждый фрагмент кода получает однозначную разметку S или D. Поливариантный BT-анализ позволяет одному и тому же участку кода иметь несколько разметок. Согласно [8] BT-анализ может быть:

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

В соответствии с данной классификацией, анализ времен связывания в JaSpe является поливаринтным по переменным, методам и классам и моноваринтным по операциям.

1.2. BT-объекты

Главными отличиями метода, реализованного в специализаторе JaSpe, от классических частичных вычислений являются принципы анализа, основанные на BT-объектах. Понятие BT-объекта впервые представлено в работах [8–15] и означает вариант разметки класса, описывающий некоторое множество экземпляров класса, порождаемых в runtime. Именно BT-объекты позволяют добиться поливариантности по классам, аналога которой нет в классическом BT-анализе. Формально BT-объект — это тройка

$$\langle TopBTType, TopTypes, FieldsAnnotation \rangle.$$

Первый компонент тройки $TopBTType$ называется *разметкой верхнего уровня* конкретного BT-объекта и также как разметка примитивного типа, принимает одно из двух значений S или D. Данный $TopBTType$. Если разметка верхнего уровня некоторого BT-объекта имеет значение S, то соответствующие данному BT-объекту экземпляры класса могут не порождаться. В таком случае данные экземпляры класса будут разделены на отдельные поля. При этом специализатор знает конкретный тип каждого из экземпляров класса, описываемых BT-объектом с S-разметкой верхнего уровня.

BT-объект, имеющий D-разметку верхнего уровня, содержит только D-поля. Это означает, что все поля примитивного типа такого BT-объекта имеют разметку D и связанные с ними операции должны присутствовать в остаточной программе. А ссылочные поля BT-объекта с D разметкой верхнего уровня также будут являться BT-объектами с D разметкой верхнего уровня. Мы будем называть такие BT-объекты D-BT-объектами. При этом специализатор не имеет информации о конкретном типе экземпляров класса, соответствующих D-BT-объекту.

Второй компонент тройки в определении BT-объекта — $TopTypes$ — множество ссылочных типов, которое называется *множество конкретных классов*, содержит классы, чью разметку описывает данный BT-объект.

И наконец $FieldsAnnotation$, третий компонент тройки — это отображение имен полей в BT-значения S или D (в случае если поле имеет примитивный тип) или BT-объекты (в случае ссылочных типов).

Отметим, что сами BT-объекты моновариантны. Это означает, если BT-анализ завершился, то в любых двух точках программы любой BT-объект имеет одну и ту же разметку.

Собственно, одна из основных идей ВТ-объектов состоит в том, как построить наиболее точные ВТ-объекты, с сохранением свойства моновариантности. Точность в данном контексте — это доля S-значений при корректной разметке программы. В свою очередь, корректность разметки для объектно-ориентированных языков изучена в работах [8, 9].

ВТ-объект в чем-то подобен реальным runtime-объектам, которые создаются в процессе исполнения программы. ВТ-объект, так же, как и runtime-объект, может содержать поля примитивного типа и ссылки на подобные ему сущности (другие ВТ-объекты). Однако, ВТ-объект — это вариант разметки класса, т. е. он соответствует множеству runtime-объектов. С другой стороны, наиболее близкой аналогией ВТ-объекта являются примитивные ВТ-значения S или D: так же как S-значение представляет из себя абстракцию для множеств числовых, логических и других значений примитивных типов, ВТ-объект — абстракция для множества различных runtime объектов, помимо прочего, утверждающая, что такие объекты похожи или не различимы на этапе ВТ-анализа в силу ограничений последнего.

1.3. Слияние ВТ-объектов

Как отмечено в конце подраздела 1.2, иногда ВТ-анализ может установить, что некоторые два ВТ-объекта похожи. Ситуации, в которых это происходит, изложены в подразделе 2.1. Также возможны случаи, когда ВТ-анализ затрудняется установить, какой из нескольких ВТ-объектов нужно использовать для дальнейшего анализа. Причины, которые приводят к последней ситуации приведены в работе [16].

Оба указанных выше случаях приводят к процедуре слияния, в результате которой из двух ВТ-объектов создается один, заменяющий первые два во всей процедуре анализа. Такое слияние может приводить к откату анализа в одно из предшествующих состояний, но с сохранением результата слияния ВТ-объектов. Такой откат необходим, поскольку ВТ-объект, являющийся результатом слияния, может иметь иную разметку, чем те из которых он получен. А значит, разметка программы основанная на исходных ВТ-объектах, существовавших до слияния, устарела и не отвечает результату слияния и замены.

Перед рассмотрением процедуры слияния ВТ-объектов, опишим слияние двух разметок примитивного типа, которое выполняется по следующему правилу: если одно из них имеет значение D, то результат будет иметь значение D, иначе результирующая разметка S.

Слияние двух BT-объектов b_0 и b_1 выполняется следующим образом:

(1) Если b_0 тот же BT-объект что и b_1, процедура завершается.
(2) Разметка верхнего уровня *TopBTType* для результата c находится по правилу для разметок примитивного типа.
(3) Если $TopBTType = D$, то b_0 и b_1 поднимаются в D.
(4) Во второй компонент тройки BT-объекта c добавляются все классы из вторых компонентов b_0 и b_1.
(5) В BT-объект c добавляются все инициализированные поля b_0.
(6) Объекты b_0 и b_1 заменяются на c в любых структурах данных, используемых анализом (в том числе во всех других BT-объектах).
(7) Происходит добавление инициализированных полей b_1 в c. Если квалифицированное имя поля из списка b_1 уже встречается в c, происходит слияние этих двух полей.

В пункте 3 может производиться поднятие BT-объектов в D. Это означает, что всем их полям примитивного типа и разметке верхнего уровня *TopBTType* присваивается D-разметка. А поля BT-объекты, соответствующие полям ссылочного типа, в свою очередь, также поднимаются в D.

При объединении двух BT-объектов могут возникнуть следующие ситуации:

- BT-объект с *TopBTType*, который был равен S, объединили с D-BT-объектом;
- изменилось разметка поля примитивного типа;
- список конкретных классов c больше, чем список у b_0 или b_1.

В этих случаях необходимо выполнить откат BT-анализа на точку создания b_0 или b_1 — того из них, который отличается в одном из пунктов от c. Если в процессе объединения выявлено несколько BT-объектов, которые вызывают откат, то откат выполняется на самую раннюю точку создания из всех рассматриваемых. Как уже отмечалось в начале данного подраздела, откат необходим, поскольку объекты моновариантны и нужно переразметить программу с обновленными BT-объектами.

2. Межпроцедурный аспект BT-анализа

В подразделах ниже излагаются основные идеи части алгоритма анализа времен связывания, ориентированной на обработку конструкций

аналогичных вызову функций (например: вызов метода, создание экземпляра класса, вызов конструктора). Данное изложение дополняет внутрипроцедурную версию алгоритма, приведенную в [16]. Некоторыми проблемами, которые решены при помощи методов, изложенных в настоящей статье, являются:

- разрешение полиморфизма в статике (без реального исполнения программы), которое не требовалось ранее, поскольку BT-анализ не выходил за рамки одного метода;
- изменение анализируемого подмножества программы в процессе BT-анализа, связанное с тем, что разметка верхнего уровня BT-объектов может стать D, а значит определить тела методов, связанных с такими BT-объектами, станет невозможным;
- решение проблемы множественного наследования от интерфейсов;
- решение проблемы рекурсии в анализируемой программе.

При анализе конструкций Java аналогичных вызову функции возможны ситуации, которые не возникали при внутрепроцедурном анализе, в том числе следующие: обработка конструкций, не являющихся ни предложениями (statement), ни выражениями (объявления полей и блоков вне тела методов); обработка вызовов, не содержащих текста (конструкторов по умолчанию); наличие множества *BT-окружений*.

BT-окружение можно определить как следующее множество:

$$Env = Env_{prim} \cup Env_{BT\text{-}Objects},$$

где Env_{prim} — множество пар вида

$$\langle ID_{var}, BT\text{-}type \rangle,$$

а $Env_{BT\text{-}objects}$ — множество пар вида

$$\langle ID_{var}, BT\text{-}object \rangle.$$

Уникальному имени переменной ID_{var} ставится в соответствие либо примитивное BT-значение S/D (в случае Env_{prim}) или BT-объект (в случае $Env_{BT\text{-}objects}$).

BT-окружение хранит текущие значения переменных и меняется в процессе анализа выражений и предложение внутри тела метода. У каждого экземпляра разметки метода свое отдельное BT-окружение.

Реализованный в JaSpe алгоритм BT-анализа интерпретирует программу аналогично методу рекурсивного спуска. При этом BT-анализ оперирует не конкретными значениями (числовыми, логическими, символьными и т. п.), а BT-значениями S, D и BT-объектами.

Алгоритм BT-анализа в JaSpe начинает интерпретацию с входной точки специализации (specialization entry point) — методе, помеченной Java-аннотацией `@Specialize`. Обнаружив входную точку, JaSpe обходит тело такого метода. При этом он строит разметку предложений и выражений на основе разметки подвыражений, а также с помощью BT-окружения и множества всех BT-объектов. В процессе построения разметки алгоритм обновляет значение переменных в BT-окружении, если обработка программной конструкции это предполагает. При объединении BT-объектов или изменении значений их полей может выполняться откат специализации (подробнее выше в подразделе 1.3).

Таким способом обрабатываются следующие конструкции: вызов метода; предложение `return`; выражения создания экземпляра класса (выражение `new`); явный вызов конструктора и явный вызов супер-конструктора. В процессе анализа этих выражений приходится также анализировать определения классов, полей, методов и блоков предложений внутри классов.

В данной работе не рассматриваются следующие конструкции: параметризованные классы и методы, методы с переменным числом аргументов, анонимные классы и лямбда выражения, вложенные классы, конструкции, связанные с рефлексией.

2.1. Вызов метода

Под вызовом метода имеются в виду конструкции следующего вида:

(*1*) `SimpleName(ArgList);`

(*2*) `Expression.SimpleName(ArgList).`

`SimpleName` в данном контексте, это идентификатор, интерпретируемый как имя метода. `ArgList` — это список аргументов, которые передается методу. `Expression` — некоторое выражение, имеющие ссылочный тип (класс или интерфейс в языке Java). Примерами такого выражения могут быть идентификатор, квалифицированное имя, доступ к полю объекта, вызов метода возвращающего экземпляр класса и т. д.

Дадим несколько определений. This BT-объект соответствует this-параметру метода. BT-сигнатура — это список BT-значений и BT-объектов, полученный для аргументов вызова метода (в порядке следования аргументов), который дополняется особенным BT-объектом — this BT-объектом — а также уникальным идентификатором имени

метода. Более формально:

$$BT\text{-}Sigature = \langle BT\text{-}ArgList, thisBTObject, MethodKey \rangle.$$

Анализ вызова метода производится по следующей общей схеме, состоящей из 6 фаз:

(*1*) Поиск this BT-объекта.

(*2*) Выяснение, поддается ли метод специализации.

(*3*) Построение множества кандидатов (разрешение полиморфизма).

(*4*) Построение множества BT-сигнатур.

(*5*) Поиск или построение разметок кандидатов на основе BT-сигнатур.

(*6*) Объединение результатов.

На первой фазе анализа вызова метода выполняется процедура поиска this BT-объекта. Для одной точки программы таких групп, как и BT-объектов, может быть несколько, поскольку возможны разные пути, по которым можно дойти до этой точки. Однако, BT-анализ устроен таким образом, что найдется не больше одного this BT-объекта, поскольку анализ рассматривает только тот путь до данной точки программы, по которому он до нее дошел в данный момент.

Чтобы найти this BT-объект анализ сначала проверяет, содержит вызываемый метод в своем определении квалификатор `static` (в языке Java такой метод называется *статическим*). В случае положительного ответа анализ считает, что для данного метода не существует this BT-объекта. Если вызываемый метод не является статическим, но содержит в своем вызове префикс вида `Expression`, то this BT-объект– это BT-объект, полученный в результате анализа `Expression`. Если оба рассмотренных выше условия не выполняются, т. е. метод не статический и у него нет префикса, то this BT-object будет тем же, что и у *экземпляра разметки метода*, в процессе построения которого встретилась данная конструкция вызова.

Экземпляр разметки метода — это структура данных, которая описывает одну из разметок метода, а также хранит вспомогательные данные, необходимые для ее построения и идентификации. Как строится экземпляр разметки метода будет изложено ниже (в рассмотрении пункта 4 обработки множества кандидатов).

Второй пункт схемы анализа вызова метода предполагает выяснение, поддается ли метод специализации. Первым шагом такой процедуры является проверка, является ли метод статическим. В

случае положительного ответа вызов метода считается поддающимся специализации. Если же метод не статический, то проверяется разметка верхнего уровня у this BT-объекта, найденного на первом шаге. В случае, если этот this BT-объект имеет S-разметку верхнего уровня, то метод считается поддающимся специализации, иначе — нет.

На третьей фазе анализа вызова метода выполняется построение множества кандидатов. Кандидат — это Java-определение метода, которое рассматривается как возможный вариант для разметки в пункте 5 схемы. На данной фазе производится разрешение полиморфизма в статике. Основой для такого разрешения является знание BT-анализа о конкретном типе runtime-объектов, чей метод может быть вызван в данной точке. Данное знание содержится в списке конкретных классов this BT-объекта. Конкретных классов может быть несколько, поэтому для каждого из них требуется найти своего кандидата. По окончанию BT-анализа, на этапе генерации остаточной программы, специализатор выберет из данных кандидатов единственный подходящий.

Множество кандидатов строится в соответствии со следующими идеями. Если метод статический, то кандидат один — тот, который определяется процедурой обработки вызова метода в спецификации Java 8 (раздел 15.12) [**17**]. Для выполнения этой процедуры JaSpe использует компилятор, встроенный в JDT [**18**].

Если же вызываемый метод не статический, то JaSpe получает *эталонную Java-сигнатуру* от JDT. Эталонная сигнатура — это сигнатура аналогичная той, которую компилятор Java строит для получения индекса метода в runtime-таблице методов (раздел 15.12 [**17**]). На основе этой эталонной сигнатуры JaSpe производит поиск определений методов, которые выше мы назвали кандидатами.

Для каждого класса из списка конкретных классов this BT-объекта находится кандидат на вызов по следующей процедуре:

- Начиная с конкретного класса и далее по классам-предкам до класса `Object`:
  - Если класс не имеет исходного кода (например, библиотечный класс), то весь вызов интерпретируется как D-вызов и анализ метода завершается.
  - Если класс содержит определение метода с эталонной Java-сигнатурой — это и есть искомое определение.
- Анализируются все интерфейсы, которые явно или неявно наследует класс, вглубь на наличие метода с квалификатором `default` и эталонной сигнатурой. Поиск выполняется пока не

будут проанализированны все интерфейсы-предки или метод не найден.

Четвертый пункт схемы анализа вызова метода предполагает построения множества BT-сигнатур. На основе анализа `ArgList` (списка выражений-аргументов метода) для каждого аргумента строится примитивное BT-значение или BT-объект.

На фазе 5 анализа вызова производится обработка множества кандидатов, в процессе которой для каждого кандидата и его BT-сигнатуры выполняется следующая процедура:

(1) Если разметка метода с такой же BT-сигнатурой уже построена ранее, то искомая разметка найдена.

(2) Если количество разметок метода превышает максимальный порог (некоторую константу), то находится ближайшая BT-сигнатура и она объединятся с текущей. Если объединение выполнено, то искомая разметка либо найдена, либо возник откат. Мерой близости двух BT-сигнатур является количество переходов из S в D, которое необходимо совершить при выполнении процедуры слияния над их аргументами. Чем меньше таких переходов, тем сигнатуры ближе.

(3) Создается новое BT-окружение для вызова. В него добавляются все пары $(ArgName, BT\text{-}Value)$, полученные из определения метода и BT-сигнатуры.

(4) Создается новый экземпляр разметки метода. Обозначим его `methodAnnotationInstance`. Он хранит стартовое BT-окружение, BT-сигнатуру, this BT-объект данного вызова и разметку возвращаемого значения (построение разметки возвращаемого значения — `BT-returnValue` — более детально рассмотренная в подразделе 2.2), которая устанавливается в S для возвращаемого значения примитивного типа или в BT-объект с полной S-разметкой для ссылочного типа. Также `methodAnnotationInstance` хранит разметку каждой конструкции в теле кандидата.

(5) `methodAnnotationInstance` добавляется в список уже проанализированных методов.

(6) Создается *D-граница*, и она добавляется в стек D-границ.

(7) Циклически анализируется тело метода-кандидата. Итерирование завершается, если разметки каждой конструкции на двух последовательных итерациях анализа тела метода-кандидата совпадают.

В процедуре анализа кандидата использовано новое понятие *D-граница*. D-граница — это пара

$$\langle T_{boundary}, P_{start} \rangle,$$

где $T_{boundary}$ — тип границы из множества {Метод, Условие}, P_{start} — позиция начала конструкции, породившей границу.

Стек D-границ используется для определения, осуществляется ли данное присваивание под D-условием. Граница типа Метод нужна для того, чтобы отличать локальные границы для локальных переменных от глобальных границ для BT-объектов.

И наконец на 6-й, последней фазе анализа вызова метода, происходит объединение результатов анализа различных кандидатов в одно BT-значение или BT-объект. После выполнение пункта 5 общей схемы, каждому кандидату поставлен в соответствие экземпляр разметки метода (`methodAnnotationInstance`), который содержит разметку результата метода (`BT-returnValue`). Анализа вызова завершается тем, что результат всех кандидатов объединяются с помощью процедуры слияния (подраздел 1.3). Такая процедура необходима, поскольку на этапе BT-анализа нельзя установит, какого из кандидатов следует выбрать. Последнее будет ясно на этапе генерации остаточной программы, когда для this BT-объекта данного вызова будет известен единственный точный ссылочный тип.

2.2. Предложение return

При исполнении предложения `return`, имеющего вид:

```
return Expression;
```

содержащий его метод завершается и возвращает в результат вычисления выражения `Expression`.

Анализ предложения `return` выполняется по следующей процедуре:

(*1*) Анализируется `Expression`. Результат анализа `Expression` — это либо примитивное BT-значение, либо BT-объект. Будем называть данный результат *текущая разметка результата метода* (текущее `BT-returnValue`).

(*2*) Проверяется, не находится ли return под D-условием. Если так, то текущее `BT-returnValue` поднимается в D. Идеи процедуры поднятия изложены в подразделе 1.3.

(*3*) Есть ли аннотация `@SpecInline` у текущего метода? Если нет, то текущее `BT-returnValue` поднимается в D. Аннотация

`@SpecInline` указывает, что если тело метода может бы подставлено вместо его вызова, то специализатор сделает это. В противном случае, если такой аннотации нет, вызов должен остаться вызовом, а значит он должен вернуть какое-то значение. Последнее в свою очередь означает, что это значение должно присутствовать в остаточной программе, т. е. должно иметь разметку D на этапе BT-анализа.

(4) Текущее `BT-returnValue` объединяется (подраздел 1.3) с полем `BT-returnValue` в соответствующей данному анализируемому методу `methodAnnotationInstance`. Результат сохраняется в оба места.

(5) Текущее `BT-returnValue` — результат разметки предложения `return`.

2.3. Выражение создания экземпляра класса

Выражение создания экземпляра класса, с учетом ограничений, изложенных в начале раздела 2 (не рассматриваются параметризированные конструкции, анонимные, а также вложенные классы), имеет вид:

```
new ClassName (ArgList).
```

`ClassName` — это имя класса, чей экземпляр создается, а `ArgList` — список аргументов, который передается конструктору экземпляра класса. Рассматриваемая конструкция предполагает, что будет создан экземпляр класса ClassName. После этого будет вызван конструктор, который выполнит необходимые, с точки зрения составителя анализируемой программы, операции по формированию пригодного для дальнейшего использования экземпляра класса.

Рассматриваемый вид выражений анализируется по следующей схеме:

(1) Создание this BT-объекта с S-разметкой верхнего уровня, все поля которого S, а конкретный класс — класс, который характеризуется именем `ClassName`.

(2) Нахождение одного (или 0) объявлений конструкторов для данного выражения. Назовем это объявление кандидатом. Поиск кандидата осуществляется в теле того класса, чей экземпляр создается. Поиск выполняется по процедуре, описанной в разделе 15.9 [17]. В данном пункте возможно, что вызываемый конструктор не имеет определения, т. е. вызывается конструктор по умолчанию. В этом случае пункт 6 данной схемы пропускается.

(*3*) Построение BT-сигнатуры. Выполняется также как и для выражения вызова метода, см. подраздел 2.1.

(*4*) Обработка предложения явного вызова другого конструктора (`this(...)`) или супер-конструктора (`super(...)`), если они присутствуют первыми предложениями в теле кандидата. Иначе обработка конструктора-предка. Конструктор в данном пункте анализируется аналогично выражению созданию экземпляра класса, но пункт 1 пропускается, т. к. при таком анализе this BT-объект заимствуется у кандидата.

(*5*) Обработка полей и блоков в декларации класса, содержащего данный конструктор. Новые поля добавляются в this BT-объект, а блоки исполняются так, как будто они находятся в теле данного конструктора.

(*6*) Построение разметки тела конструктора на основе BT-сигнатуры. Выполняется также как и пункт 5 ("поиск или построение разметок кандидатов на основе BT-сигнатур") в выражении вызова метода (см. подраздел 2.1), но для одного кандидата, которые был найден в пункте 2.

2.4. Вызов (супер-)конструктора

Предложения данного вида имеют вид:

(*1*) `super(ArgList);`

(*2*) `this(ArgList);`

Здесь `super` и `this` — ключевые слова, означающие вызов конструктора родительского класса (`super`) или данного класса (`this`). `ArgList` — как и ранее, список аргументов вызова. Рассматриваемые предложения могут встречаться только в начале тела какого-либо конструктора. Анализ данных конструкций аналогичен анализу создания экземпляра класса, где пункт 1 опускается, поскольку this BT-объект заимствуется у конструктора, осуществляющего данные вызовы.

3. Пример анализа Java-программы

В качестве примера рассмотрим программу представленную на рисунке 1. В ней определено четыре класса:

- `MethodExample` — основной класс, который содержит определение метода `example`. Метод `example` — это главный содержательный метод.

```
1  public class MethodExample {
2      @Specialize
3      Number example(double dArg) {
4          boolean S = true;
5          Number obj, res;
6
7          if (S) {
8              obj = new ErshovNumber(1.0);
9          } else {
10             obj = new DummyNumber(dArg);
11         }
12         res = obj.pow(3).pow(2);
13         return res;
14     }
15 }
16 public class Number {
17     double val;
18
19     public Number(double x) {
20         val = x;
21     }
22     @SpecInline
23     Number pow (int n) {
24         return new Number(Math.pow(val, n));
25     }
26 }
```

```
27 public class ErshovNumber extends Number{
28     ErshovNumber(double x) { super(x); }
29
30     @SpecInline
31     Number pow (int n) {
32         double r = 1, x = val;
33         while(n != 0)
34             if (n%2 == 0) {
35                 x = x * x; n = n / 2;
36             } else {
37                 r = r * x; n = n - 1;
38             }
39         return new ErshovNumber(r);
40     }
41 }
42 public class DummyNumber extends Number{
43     DummyNumber(double x) { super(x); }
44 }
```

Рисунок 1. Пример разметки

- `Number` — базовый класс, определяющий сущность “число”, экземпляр которого создается конструктором `Number` и возводится в степень с помощью метода `pow`. Этот метод `pow` выполняет свою функцию на основе библиотечного метода `Math.pow`.
- `ErshovNumber` — класс, унаследованный от класса `Number`, который переопределяет метод `pow`. Новый вариант метода `pow`, возводит число в степень на основе ершовского алгоритма.
- `DummyNumber` — класс, являющийся потомком `Number`, который не добавляет никакой новой функциональности, а служит для иллюстрации.

Метод с именем example, являющийся главным в данной программе, в зависимости от значения константы S, определенной в строке 4, создает один из двух экземпляров класса: либо `ErshovNumber` (строка 8), либо `DummyNumber` (строка 10). Конструктору `ErshovNumber` передается известное на стадии специализации число `1.0`, а конструктору `DummyNumber` — неизвестное `dArg`. Далее, переменной `res` присваивается ссылка на один из экземпляров класса, полученный на основе чисел, переданных конструкторам, путем возведения этих чисел сначала в степень `3`, затем в степень `2` (строка 12). Экземпляр класса, сохраненный в переменной `res`, возвращается в качестве результат работы метода `example` (строка 13).

Зеленым фоном на рисунке 1 отмечены S-конструкции, прозрачными прямоугольниками обведены D-конструкции, а конструкции, получившие оба варианта разметки в силу поливариантности, подчеркнуты волнистой линией.

Рассмотрим процесс BT-анализа метода `example`. Сначала строится разметка аргументов: рассматриваемый метод имеет один аргумент `dArg`, который получает D-разметку, поскольку его значение неизвестно. Далее анализируются объявления переменных `S`, `obj` и `res` (строки 4 и 5), которые получают S-разметку, поскольку их значения либо очевидны, либо переменным не присвоены никакие значения. Затем обрабатывается предложение `if` (строки 7-11). В процессе обработки предложения `if` анализируются выражения создания экземпляра классов `ErshovNumber` и `DummyNumber`. Мы подробнее остановимся на разборе второго случая. Отметим лишь, что анализ выражения `new ErshovNumber(1.0)` порождает BT-объект с S разметкой верхнего уровня. В списке конкретных классов этого BT-объекта находится один класс `ErshovNumber`, а поле `val` имеет S-разметку. Обозначим этот BT-объект как a.

Анализ выражения `new DummyNumber(dArg)` выполняется по схеме, приведенной в подразделе 2.3:

(*1*) Создается новый BT-объект с S-разметкой верхнего уровня, конкретным классом `DummyNumber`, и S-полем `val`. Обозначим этот BT-объект как b.

(*2*) Конструктором, анализ которого необходимо произвести, является конструктор находящийся в определении `DummyNumber` и имеющем заголовок `DummyNumber(double x)`.

(*3*) Строится BT-сигнатура:

$$BT\text{-}Signature_1 = \langle (D), b, DummyNumber.DummyNumber(double) \rangle.$$

(*4*) В теле кандидата `DummyNumber(double)` присутствует явный вызов супер-конструктора. Это предложение выполняется аналогично данному примеру. В качестве кандидата супер-конструктора выступает метод `Number(double x)` в классе `Number`. Анализ этого кандидата во многом аналогичен данному примеру с одним существенным исключением: в теле супер-конструктора выполняется присваивание в поле `val` D-значения, поскольку разметка `dArg` равна D.

(*5*) Объявление класса `DummyNumber` не содержит полей и блоков, поэтому никаких действий в этом пункте выполнять не требуется.

(*6*) Опишем построение разметки кандидата `DummyNumber(double)`. Ранее разметка данного конструктора с BT-сигнатурой

$$BT\text{-}Signature_1 = \langle (D), b, DummyNumber.DummyNumber(double) \rangle$$

не строилась. Поскольку это первая разметка конструктора `DummyNumber(double)`, то количество его разметок не превышает максимальный порог. Создается новое ВТ-окружение

$$Env_1 = \{(DummyNumber.DummyNumber(double).x, S)\}.$$

Обратите внимание, что для аргумента `x` использовано квалифицированное имя, позволяющее отличить его от любого другого имени в программе. Теперь создается `methodAnnotationInstance`, который хранит Env_1, ВТ-signature$_1$, b. `BT-returnValue` можно не создавать, поскольку конструктор не возвращает результат. Экземпляр разметки метода `methodAnnotationInstance` добавляется в список уже проанализированных методов. Создается D-граница {43, Method}, где 43 — это строка в которой начинается определение конструктора. D-граница добавляется в стек D-границ. Строка в качестве начальной позиции достаточна для данного примера, однако, возможны более точные границы, например, позиция в файле символа начала определения. Тело конструктора содержит одно предложение `super(...)`, которая уже проанализирована, поэтому никаких дополнительных действий выполнять не требуется.

После анализа тела предложения `if` (строки 7–11) ВТ-анализ выполняет слияние состояний ВТ-окружения, полученных в результате анализа веток `then` и `else`. Необходимость этого слияния обсуждается в статье [16] подразделе 4.2. В результате такого слияния ВТ-объекты a, полученный в строке 8, и b, полученный в строке 10, будут заменены на ВТ-объект c. Причем ВТ-объект c имеет S-разметку верхнего уровня, множество конкретных классов ВТ-объекта c состоит из двух элементов — `ErshovNumber` и `DummyNumber`, а поле `val` имеет D-разметку. После этого происходит откат специализации на строку 8 — место создания ВТ-объекта a, поскольку разметка его поля изменилась с S на D. Далее анализ обрабатывает выражения создания экземпляров класса и связанные с ними присваивания. Фактически анализ повторяет свои шаги, только в качестве this ВТ-объекта будет выступать ВТ-объект c.

В итоге специализатор снова завершит ВТ-анализ веток `then` и `else` предложения `if` и выясниться, что оба состояния ВТ-окружения содержат пару (obj, c), поэтому процедура слияния завершится на 1-м шаге, без отката. Теперь ВТ-анализ переходит к строке 12, входит в присваивание и, начиная анализировать его правую часть, встречает выражение вызова метода — `obj.pow(3).pow(2)`. Это составное выражение, которое включает вызов метода `obj.pow(3)` и затем производится

вызов метода `pow(2)` у объекта, полученного в качестве результата предыдущего вызова.

Рассмотрим как обрабатывается вызов `obj.pow(3)`. Анализ этого метода производится в соответствии со схемой, представленной в начале подраздела 2.1:

(*1*) Определяется this BT-объект: для этого анализируется выражение, представляющее из себя имя переменной `obj`. Результат, BT-объект c, будет прочитан из BT-окружения.

(*2*) Данный метод поддается специализации, он не статический, но BT-объект c имеет S-разметку верхнего уровня.

(*3*) Для каждого из конкретных классов `ErshovNumber` и `DummyNumber` находятся кандидаты на анализ. Эталонная сигнатура будет `Number pow (int)`. Для конкретного класса `ErshovNumber` кандидатом будет являться метод `ErshovNumber.pow(int)`, для `DummyNumber` — `Number.pow(int)`.

(*4*) Строятся BT-сигнатура для каждого из кандидатов: кандидату `ErshovNumber.pow(int)` соответствует BT-сигнатура

$$BT\text{-}Sigature_2 = \langle (S), c, ErshovNumber.pow(int\ n) \rangle.$$

Кандидату `Number.pow(int)` соответствует BT-сигнатура

$$BT\text{-}Sigature_3 = \langle (S), c, Number.pow(int\ n) \rangle.$$

(*5*) Для каждого из кандидатов разметка строится аналогично пункту 6 из примера для создания экземпляра класса `DummyNumber`.

(*6*) На этом шаге производится объединение результатов кандидатов, т. е. BT-объектов пораженных при анализе строк 24 и 39. Результирующим будет BT-объект

$$d = \langle S, \{ErshovNumber, Number\}, \{(Number.val, D)\} \rangle.$$

Теперь производится анализ вызова метода `obj.pow(3).pow(2)`, который производится аналогично вызову `obj.pow(3)` с той особенностью, что this BT-объектов в данном случае является BT-объект d. Результатом будет является BT-объект

$$e = \langle S, \{ErshovNumber, Number\}, \{(Number.val, D)\} \rangle.$$

После обработки последнего вызова метода в примере производится присваивание в переменную `res`, т. е. в BT-окружение помещается пара $(\texttt{res}, e)$. Ради простоты изложения мы использовали простое имя `res` в качестве идентификатора в паре BT-окружения. В более сложных случаях разумно использовать более сложные квалифицированные имена.

И наконец, процедура анализа достигает последней 13 строки метода `example`. В этой строке производится анализ предложения `return`. Поскольку метод `example` является входной точкой специализации (и потому не содержит аннотации `@SpecInline`), BT-объект e поднимается в D, т. е. все его поля и разметка верхнего уровня устанавливаются в D. После чего происходит откат на точку создания этого BT-объекта (поскольку e имел S-разметку верхнего уровня) — на строку 39 процесса анализа вызова метода `obj.pow(3).pow(2)`. В итоге анализа вызова последнего завершится и результат работы этого метода получит в качестве разметки BT-объект

$$e = \langle D, \{ErshovNumber, Number\}, \{(Number.val, D)\}\rangle.$$

Затем, повторяя анализ присваивания в строке 12, BT-объект e будет помещен в BT-окружение в составе пары $(\texttt{res}, e)$. И в завершении анализа метода `example` будет проанализировано предложение `return`, в результате чего BT-объект e будет помещен в `BT-returnValue` экземпляра разметки метода `example`. На этом анализ завершается.

4. Другие работы и обсуждение

В данной статье излагаются методы, которые получены на основе адаптации уже имеющихся алгоритмов анализа времен связывания, путем их расширения до поливариантности по классам и распространения на объектно-ориентированный язык общего назначения.

Предшественником JaSpe среди специализаторов программ на императивных языках является Tempo [19]. Tempo разработан в лаборатории IRISA/INRIA Университета Ренна (Франция) и работает методом частичных вычислений, специализируя программы на языке C. Tempo, как и JaSpe, использует offline-стратегию частичных вычислений, однако выполняет более двух проходов по программе. К дополнительным проходам относятся анализ синонимов (alias analysis) и анализ определений (definition analysis), которые решают проблемы, связанные с наличием указателей в языке C. BT-анализ в Tempo поливариантен по переменным и методам, но моновариантен по структурам, аналогу классов в императивном программировании. В отличии от Tempo BT-анализ в специализаторе JaSpe, которому посвящена настоящая статья, поливариантен по классам. Поливариантность по классам особенно важна при анализе времен связывания программ на объектно-ориентированных языках,

поскольку те предполагают создание большого количества экземпляров класса, которые имеют разное назначение и поэтому важно, чтобы они размечались по-разному. Последнее позволяет увеличить число оптимизаций в таких программах.

На основе Tempo Ульриком Шульцем в Университете Южной Дании был разработан специализатор JSpec [**20**], работающий с программами на языке Java. Особенностью JSpec является то, что он транслирует программу на Java в C, а затем специализирует ее при помощи Tempo. Это означает, что JSpec наследует свойства Tempo, т. е. ВТ-анализ, который он использует, моновариантен по классам. Моновариантность по классам препятствует практическому применению JSpec. В специализаторе JaSpe это ограничение преодолено, что приближает частичные вычисления объектно-ориентированных программ к практике.

Специализатором наиболее близким по идеям к JaSpe является частичный вычислитель CILPE [**8**–**15**], использующий offline-стратегию. CILPE был разработан Юрием Климовым в ИПМ им.М.В.Келдыша РАН (Москва). Специализатор CILPE предназначен для применения к программам на подмножестве стекового объектно-ориентированного языка CIL, байт-кода платформы Microsoft .NET. Особенностью этого подмножества языка CIL является то, что оно содержит небольшое число различных конструкций, во многом ориентированных на работу со стеком. В специализаторе CILPE впервые были использованы ВТ-объекты. Однако, методы анализа заложенные в CILPE пришлось адаптировать для применения в JaSpe в силу специфики подмножества CIL. Также, в отличие от CILPE, JaSpe ориентирован на интерактивное взаимодействие с пользователем. Последнее внесло некоторые различия в использованных методах.

Подход, названный его авторами гибридными частичными вычислениями, реализуется в специализаторе Civet [**21**]. Суть этого подхода состоит в том, что программист-пользователь самостоятельно указывает, какие части кода известны на стадии специализации. Среди оставшейся части кода выделяются конструкции, которые зависят только от известных конструкций, и они также считаются известными. На основании этой информации с помощью частичных вычислений, работающих по online-стратегии, строится специализированная версия программы. Специализатор JaSpe, напротив, использует offline-стратегию частичных вычислений, т. е. выполняет два прохода. Этап генерации остаточной программы следует за этапом анализом

времен связывания, который в отличии от ручной разметки в Civet автоматически выделяет известные и неизвестные конструкции в программе. Это освобождает программиста от ненужной работы.

Другим специализатором использующим нестандартный подход является PE-KeY [22]. Он основан на среде верификации KeY и системе правил для этой среды. PE-KeY работает в две стадии. На первой стадии происходит символьное выполнение программы и ее частичное вычисление. На второй стадии генерируется остаточная программа. Такой подход близок к частичным вычислениям, работающим по offline-стратегии, в том числе и к подходу использованном в нашем специализаторе JaSpe. Однако, PE-KeY использует символьное выполнение на первой стадии, в то время как первая стадия в JaSpe основывается на BT-анализе. Также, PE-KeY не поддерживает операции над числами с плавающей точкой из-за свойств системы верификации KeY, лежащей в его основе. Напротив, в JaSpe нет ограничений, связанных с операциями над данными примитивных типов.

В последнее время появился ряд работ, которые посвященны практическому применению частичных вычислений.

Во-первых, необходимо упомянуть современную разработку — GraalVM [23, 24], в которой реализуется поддержка многоязыковых интерпретаторов. GraalVM содержит в своем составе специализатор, работающий методом частичных вычислений. Этот специализатор, опираясь подсказки пользователя, генерирует специализированную версию фрагментов программы в runtime. Такая специализация позволяет повысить производительность. Эффективность такого подхода согласно [23] была опробована на интерпретаторах языков JavaScript, Ruby и R. Также отметим, что согласно [25] GraalVM поддерживает компиляцию перед исполнением (ahead-of-time compilation). Это позволяет значительно снизить время на запуск программ.

Авторы [26] представили алгоритм CompGen, который сокращает время JIT-компиляции в GraalVM. Этот алгоритм основывается на идеи генерации компилятора по интерпретатору. В основе лежит достаточно старая идея, которая сформулирована как вторая проекция Футамуры [27]. Compgen применяется не ко всему специализатору, а только к некоторой его части, ответственной за оптимизацию определенных узлов. В статье [26] узлы выбирались на основе статистики, собранной при профилировании. В целом работа [26] показывает, что вторая проекция Футамуры может иметь успешное практическое применение.

В статье [28] обсуждается трансляция программ с языка функционального программирование Gallina в язык C. Эта трансляция использет частичные вычисления. Язык Gallina применяется в системе доказательства теорем Coq. К особенностям этого языка, которых нет в C, относятся, помимо прочего, полиморфизм типов и зависимые типы. За устранение этих двух особенностей из программного кода и отвечают частичные вычисления. Данное применение частичных вычислений интересно, поскольку позволяет создавать все более высокоуровневые языки, сохраняя возможность их трансляции в низкоуровневые, с возможными оптимизациями.

Задача выравнивания последовательностей ДНК решается в работе [29] с использованием метода частичных вычислений. В этой работе алгоритм библиотеки, которую авторы называли AnySeq, разделен на две части: общую часть и часть, предназначенную для оптимизации под различные архитектуры. Применение частичных вычислений позволяет эффективно удалить накладные расходы от использования высокоуровневых абстракций. Полученные результаты применения AnySeq сопоставимы с результатами современных оптимизированных вручную библиотек выравнивания последовательностей ДНК. При этом AnySeq может работать как с процессорами общего назначения, так и с графическими картами (graphics processing unit, GPU) и с ПЛИС. Обсуждаемая работа показывает, что используя общий подход в программировании, при помощи частичных вычислений можно получить эффективный результат с возможностью его быстрого переноса на различные архитектуры.

Упомянем две работы, связанные с частичными вычислениями программ для GPU. Первая из них [30] посвящена рендеренгу сцен в компьютерной анимации. В [30] дается описание подхода, на основе которого построена библиотека Rodent, которая использует частичные вычисления для оптимизации шейдеров, в случае если известна модель сцены. Такой подход позволяет получать выигрышь до 40%, что является значительной величиной, учитывая какое время может затрачиваться на расчет реалистичной компютерной анимации для практических приложений.

В другой работе [31] частичные вычисления также были пременены к программам для GPU. В ней рассматривалась задача извлечения файлов из необработанных данных. Общая программа специализировалась по строке-шаблону, по которому производится поиск. Результирующая программа исполнялась на GPU. Благодаря частичным вычислениями

было получено ускорение до восьми раз. Такое достаточно большое ускорение позволяет надеяться, что частичные вычисления могут получить применение и в области вычислений на GPU.

Последние три работы, рассмотренные выше, выполнены на основе системы программирования AnyDSL [**32**]. В состав AnyDSL входит специализатор, который позволяет легко применять частичные вычисления к программам на языке данной системы. Этот специализатор использует online-стратегию.

Заключение

В статье были рассмотрены методы поливариантного анализа времен связывания таких конструкций объектно-ориентиро-ванного языка как вызовы методов, выражения создания экземпляров класса и явные вызовы конструкторов. Работа алгоритма анализа времен связывания, основанного на BT-объектах, была проиллюстрирована его применением к примеру программы на языке Java. Сформулированные в настоящей статье методы расширяют алгоритм внутрипроцедурного анализа, приведенный в [**16**], и увеличивают силу последнего, позволяя проводить большее число оптимизаций.

Возможны следующие направления развития рассмотренных методов, связанные с распространием их на следующие конструкции языка Java:

- параметризованные классы и методы;
- методы с переменным числом аргументов;
- анонимные классы и лямбда выражения;
- вложенные классы.

Исполняемая программа. Представленные в данной статье методы реализованы в алгоритме BT-анализа, который доступен в виде набора плагинов для Eclipse IDE по адресу:

ftp://ftp.botik.ru/rented/iaadamovich/JaSpe_BTA/.

Благодарности

Автор выражает особую благодарность Андрею Валентиновичу и Юрию Андреевичу Климовым за консультации по методу частичных вычислений, ценные советы, а также конструктивную критику решений, рассмотренных при реализации специализатора JaSpe.

Список литературы

[1] V. F. Turchin. “The Concept of a Supercompiler,”, *ACM Transactions on Programming Languages and Systems*, **8**:3 (1986), pp. 292-325. ↑4

[2] 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. D. Bjørner, M. Broy, I. V. Pottosin, Springer, ISBN 978-3-540-49637-3, pp. 227-248. ↑4

[3] Анд. В. Климов, *Введение в метавычисления и суперкомпиляцию*, КомКнига, М., 2008, ISBN 978-5-484-01028-8, с. 343–368. ↑4

[4] Анд. В. Климов, С. А. Романенко. «Суперкомпиляция: основные принципы и базовые понятия», *Препринты ИПМ им. М.В. Келдыша*, 2018, 111, 36 с. ↑4

[5] 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. ↑4,5

[6] *Eclipse integrated develoment environment (IDE)*, Eclipse Foundation. ↑4

[7] N. H. Christensen, R. Glück. “Offline partial evaluation can be as accurate as online partial evaluation”, *ACM Transactions on Programming Languages and Systems*, **26**:1 (2004), pp. 91–220. ↑5

[8] 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. ↑6,7,8,23

[9] Ю. А. Климов. «Специализатор CILPE: анализ времен связывания», *Препринты ИПМ им. М.В. Келдыша*, 2009, 007, 28 с. ↑7,8,23

[10] 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. ↑7,23

[11] 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. ↑7,23

[12] 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. ↑7,23

[13] Ю. А. Климов. «Специализатор CILPE: генерация остаточной программы», *Препринты ИПМ им. М.В. Келдыша*, 2009, 008, 26 с. ↑7,23

[14] Ю. А. Климов. «Специализатор CILPE: доказательство корректности», *Препринты ИПМ им. М.В. Келдыша*, 2009, 033, 32 с. ↑7,23

[15] Ю. А. Климов. «Специализатор CILPE: частичные вычисления для объектно-ориентированных языков», *Программные системы: теория и приложения : электрон. научн. журн.*, 2010, №3(3), с. 13–36. ↑7,23

[16] И. А. Адамович, Ю. А. Климов. «Специализатор JaSpe: алгоритм внутрипроцедурного анализа времени связывания программ на подмножестве языка Java», *Программные системы: теория и приложения*, **11**:1(44) (2020), с. 3-–29. ↑8,10,20,26

[17] J. Gosling, B. Joy, G. L. Steele, G. Bracha, A. Buckley. *The Java® language specification*, Java SE 8 Edition, Addison-Wesley Professional, 2014, ISBN 978-0-13-390069-9. ↑13,16

[18] *Eclipse Java development tools (JDT)*, Eclipse Foundation. ↑13

[19] C. Consel, J. L. Lawall, A.-F. Le Meur. "A tour of Tempo: a program specializer for the C language", *Science of computer programming*, **52**:1–3 (2004), pp. 241-370. ↑22

[20] U.P. Schultz. *Object-oriented software engineering using partial evaluation*, Ph.D. dissertation, University of Rennes I, Rennes, France, 2000. ↑23

[21] A. Shali, W. R. Cook. "Hybrid partial evaluation", *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, Association for Computing Machinery, New York, NY, USA, 2011, ISBN 9781450309400, pp. 375–390. ↑23

[22] R. Ji, R. Bubel. "PE-KeY: A partial evaluator for Java programs", Lecture Notes in Computer Science, eds. J. Derrick, S. Gnesi, D. Latella, H. Treharne, Springer, Berlin–Heidelberg, 2012, ISBN 978-3-642-30728-7, pp. 283–295. ↑24

[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. ↑24

[24] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, M. Wolczko. "One VM to rule them all", *Onward! 2013: Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software* (October 2013), pp. 187–204. ↑24

[25] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, T. Würthinger. "Initialize once, start fast: application initialization at build time", *Proceedings of the ACM on Programming Languages*. V. 3 (October 2019), 2019, 29 pp. ↑24

[26] F. Latifi, D. Leopoldseder, C. Wimmer, H. Mössenböck. "CompGen: generation of fast JIT compilers in a multi-language VM", *Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS '21 (October 2021), 2021, pp. 35–47. ↑24

[27] Y. Futamura. "Partial evaluation of computation process — an approach to a compiler-compiler", *Higher-Order and Symbolic Computation*, **12**:4 (1999), pp. 381–391. ↑24

[28] A. Tanaka. “Coq to C translation with partial evaluation”, *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2021 (January 2021), pp. 14–31. ↑25

[29] A. Müller, B. Schmidt, A. Hildebrandt, R. Membarth, R. Leißa, M. Kruse, S. Hack. “AnySeq: A high performance sequence alignment library based on partial evaluation”, IEEE International Parallel and Distributed Processing Symposium (IPDPS) (18–22 May 2020), pp. 1030–1040. ↑25

[30] A. Pérard-Gayot, R. Membarth, R. Leißa, S. Hack, P. Slusallek. “Rodent: generating renderers without writing a generator”, *ACM Transactions on Graphics*, **38**:4 (2019), 40, 12 pp. ↑25

[31] A. Tyurin, D. Berezun, S. Grigorev. “Optimizing GPU programs by partial evaluation”, PPoPP ’20: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (February 2020), 2020, pp. 431–432. ↑25

[32] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. MÏler, B. Schmidt. “AnyDSL: a partial evaluation framework for programming high-performance libraries”, *Proceedings of the ACM on Programming Languages*, **2**:OOPSLA (November 2018), 119, 30 pp. ↑26

Поступила в редакцию 08.11.2021
Переработана 19.11.2021
Опубликована 01.12.2021

Пример ссылки на эту публикацию:

И. А. Адамович. «Специализатор JaSpe: BT-объекты и межпроцедурный аспект алгоритма анализа времен связывания». *Программные системы: теория и приложения*, 2021, **12**:4(51), с. 3–32.
10.25209/2079-3316-2021-12-4-3-32
http://psta.psiras.ru/read/psta2021_4_3-32.pdf

Об авторе:

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

CSCSTI 50.05.13
UDC 519.681.3

Igor A. Adamovich. *The JaSpe specializer: BT-objects and the interprocedural aspect of the binding-time analysis algorithm.*

ABSTRACT. This paper is devoted to partial evaluations that use the offline strategy. The power of this method for solving the problem of program specialization depends mainly on the binding-time analysis, which annotates the program constructs as either executable or not executable at the stage of specialization.

The binding-time analysis can use several variants of the annotation of the class fields, depending on their use in the program. Increasing the number of additional optimizations, this class polyvariance allows more programs to be specialized effectively. The better effect is achieved during the specialization of object-oriented programs, involving the creation of many class instances with different purposes.

Known algorithms for the binding-time analysis are extended to be class polyvariant and applied to a general-purpose, object-oriented language. The new methods are implemented as plugins for the Eclipse IDE. The plugins form the JaSpe specializer for Java programs.

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

2020 *Mathematics Subject Classification*: 68N15; 68N17,68N18

References

[1] V. F. Turchin. "The Concept of a Supercompiler,", *ACM Transactions on Programming Languages and Systems*, **8**:3 (1986), pp. 292-325. ↑4

[2] 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. D. Bjørner, M. Broy, I. V. Pottosin, Springer, ISBN 978-3-540-49637-3, pp. 227-248. ↑4

[3] And. V. Klimov, *Introduction to metacomputations and supercompilation*, KomKniga, M., 2008, ISBN 978-5-484-01028-8, pp. 343–368 (In Russian). ↑4

[4] And. V. Klimov, S. A. Romanenko. "Supercompilation: fundamentals and basic concepts", *Preprinty IPM im. M.V. Keldysha*, 2018, 111 (In Russian), 36 pp. ↑4

[5] 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. ↑4,5

[6] *Eclipse integrated develoment environment (IDE)*, Eclipse Foundation. ↑4

© I. A. ADAMOVICH, 2021
© AILAMAZYAN PROGRAM SYSTEMS INSTITUTE OF RAS, 2021
© PROGRAM SYSTEMS: THEORY AND APPLICATIONS (DESIGN), 2021

10.25209/2079-3316-2021-12-4-3-32

[7] N. H. Christensen, R. Glück. "Offline partial evaluation can be as accurate as online partial evaluation", *ACM Transactions on Programming Languages and Systems*, **26**:1 (2004), pp. 91–220. ↑5

[8] 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). ↑6,7,8,23

[9] Yu. A. Klimov. "Specializer CILPE: binding time analysis", *Preprinty IPM im. M.V. Keldysha*, 2009, 007 (In Russian), 28 pp. ↑7,8,23

[10] 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. ↑7,23

[11] 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. ↑7,23

[12] Yu. A. Klimov. "Specializer CILPE: examples of object-oriented program specialization", *Preprinty IPM im. M.V. Keldysha*, 2008, 030 (In Russian), 28 pp. ↑7,23

[13] Yu. A. Klimov. "Specializer CILPE: residual program generation", *Preprinty IPM im. M.V. Keldysha*, 2009, 008 (In Russian), 26 pp. ↑7,23

[14] Yu. A. Klimov. "Specializer CILPE: correctness proof", *Preprinty IPM im. M.V. Keldysha*, 2009, 033 (In Russian), 32 pp. ↑7,23

[15] Yu. A. Klimov. "Specializer CILPE: partial evaluator for object-oriented languages", *Program Systems: Theory and Applications : elektron. nauchn. zhurn.*, 2010, no. 3(3), pp. 13–36 (In Russian). ↑7,23

[16] 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). ↑8,10,20,26

[17] J. Gosling, B. Joy, G. L. Steele, G. Bracha, A. Buckley. *The Java® language specification*, Java SE 8 Edition, Addison-Wesley Professional, 2014, ISBN 978-0-13-390069-9. ↑13,16

[18] *Eclipse Java development tools (JDT)*, Eclipse Foundation. ↑13

[19] C. Consel, J. L. Lawall, A.-F. Le Meur. "A tour of Tempo: a program specializer for the C language", *Science of computer programming*, **52**:1–3 (2004), pp. 241-370. ↑22

[20] U.P. Schultz. *Object-oriented software engineering using partial evaluation*, Ph.D. dissertation, University of Rennes I, Rennes, France, 2000. ↑23

[21] A. Shali, W. R. Cook. "Hybrid partial evaluation", *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, Association for Computing Machinery, New York, NY, USA, 2011, ISBN 9781450309400, pp. 375–390. ↑23

[22] R. Ji, R. Bubel. "PE-KeY: A partial evaluator for Java programs", Lecture Notes in Computer Science, eds. J. Derrick, S. Gnesi, D. Latella, H. Treharne, Springer, Berlin–Heidelberg, 2012, ISBN 978-3-642-30728-7, pp. 283–295. ↑24

[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. ↑24

[24] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, G. Duboscq, C.

Humer, G. Richards, D. Simon, M. Wolczko. “One VM to rule them all”, *Onward! 2013: Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software* (October 2013), pp. 187–204. ↑24

[25] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, T. Würthinger. “Initialize once, start fast: application initialization at build time”, *Proceedings of the ACM on Programming Languages*. V. 3 (October 2019), 2019, 29 pp. ↑24

[26] F. Latifi, D. Leopoldseder, C. Wimmer, H. Mössenbëk. “CompGen: generation of fast JIT compilers in a multi-language VM”, *Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS ’21 (October 2021), 2021, pp. 35–47. ↑24

[27] Y. Futamura. “Partial evaluation of computation process — an approach to a compiler-compiler”, *Higher-Order and Symbolic Computation*, **12**:4 (1999), pp. 381–391. ↑24

[28] A. Tanaka. “Coq to C translation with partial evaluation”, *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2021 (January 2021), pp. 14–31. ↑25

[29] A. Müller, B. Schmidt, A. Hildebrandt, R. Membarth, R. Leißa, M. Kruse, S. Hack. “AnySeq: A high performance sequence alignment library based on partial evaluation”, IEEE International Parallel and Distributed Processing Symposium (IPDPS) (18–22 May 2020), pp. 1030–1040. ↑25

[30] A. Pérard-Gayot, R. Membarth, R. Leißa, S. Hack, P. Slusallek. “Rodent: generating renderers without writing a generator”, *ACM Transactions on Graphics*, **38**:4 (2019), 40, 12 pp. ↑25

[31] A. Tyurin, D. Berezun, S. Grigorev. “Optimizing GPU programs by partial evaluation”, PPoPP ’20: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (February 2020), 2020, pp. 431–432. ↑25

[32] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. MLler, B. Schmidt. “AnyDSL: a partial evaluation framework for programming high-performance libraries”, *Proceedings of the ACM on Programming Languages*, **2**:OOPSLA (November 2018), 119, 30 pp. ↑26

Sample citation of this publication:

Igor A. Adamovich. “The JaSpe specializer: BT-objects and the interprocedural aspect of the binding-time analysis algorithm”. *Program Systems: Theory and Applications*, 2021, **12**:4(51), pp. 3–32. (*In Russian*). 10.25209/2079-3316-2021-12-4-3-32

http://psta.psiras.ru/read/psta2021_4_3-32.pdf