На правах рукописи

Климов Юрий Андреевич

Специализация программ на объектно-ориентированных языках методом частичных вычислений

Специальность 05.13.11 — Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Автореферат
диссертации на соискание ученой степени
кандидата физико-математических наук

Москва – 2009

Работа выполнена в Институте прикладной математики им. М.В. Келдыша РАН.

Научный руководитель:	кандидат физико-математических наук Романенко Сергей Анатольевич
Официальные оппоненты:	доктор физико-математических наук, старший научный сотрудник Петренко Александр Константинович
	кандидат физико-математических наук, старший научный сотрудник Бульонков Михаил Алексеевич
Ведущая организация:	Институт программных систем им. А.К. Айламазяна РАН

Защита состоится 17 ноября 2009 г. в 11 часов на заседании Диссертационного совета Д 002.024.01 при Институте прикладной математики им. М.В. Келдыша РАН по адресу: 125047, Москва, Миусская пл., 4.

С диссертацией можно ознакомиться в библиотеке Института прикладной математики им. М.В. Келдыша РАН.

Автореферат разослан 14 октября 2009 г.

Ученый секретарь диссертационного совета,
доктор физико-математических наук Т.А. Полилова

Общая характеристика работы

Объект исследования и актуальность работы

Сложность решаемых с помощью компьютера задач, а также затрачиваемое время на их реализацию постоянно возрастают. Для облегчения работы программиста используются различные методы: автоматическое управление памятью, наборы стандартных библиотек, проблемно-ориентированные языки программирования и т.п. Данные методы, с одной стороны, значительно облегчают труд программиста. С другой стороны, эффективность программ при их использовании может значительно снижаться по сравнению с полностью ручной оптимальной и эффективной реализацией, создание которой, однако, может занимать слишком много времени.

В данной работе исследуется проблема оптимизации и преобразования программ на объектно-ориентированных языках программирования на основе априорной информации об аргументах программ или контексте использования объектов и методов (подпрограмм).

Подход к оптимизации программ, основанный на учете дополнительной информации об условиях, в которых будет эксплуатироваться программа, называется *специализацией* программ. В частности, в качестве таких ограничений могут использоваться зафиксированные значения некоторой части исходных данных, не изменяющиеся от одного запуска программы к другому. Или, в общем случае, ограничения на исходные данные могут быть заданы в виде условий, выраженных на некотором языке спецификаций. В таких случаях принято говорить, что выполняется специализация программы по отношению *к ограничениям на исходные данные*.

Методы специализации изначально развивались для функциональных языков, и в этом направлении был достигнут существенный прогресс, причем наиболее широко распространенным подходом является метод *частичных вычислений* (Partial Evaluation, PE).

В настоящее время получили очень широкое распространение объектно-ориентированные языки программирования, такие как C# и Java. Однако методы специализации для объектно-ориентированных языков развиты в гораздо меньшей степени, чем для функциональных языков.

Цель и задачи работы

Целью работы является исследование и разработка основанных на частичных вычислениях методов и алгоритмов для специализации программ на объектно-ориентированных языках программирования, таких как C# и Java. А также проверка работоспособности этих алгоритмов путем создания экспериментального специализатора.

Основные задачи работы:

- Исследование существующих методов частичных вычислений для специализации объектно-ориентированных программ.
- Разработка метода частичных вычислений, обладающего поливариантностью и обеспечивающего повышение эффективности программ за счет выполнения части операций во время специализации, удаления части объектов и изменения представления данных, для полного объектно-ориентированного языка.
- Реализация разработанных методов и алгоритмов в экспериментальном специализаторе и их апробация на модельных задачах.

Научная новизна работы

В работе формально описан новый метод частичных вычислений для объектно-ориентированных языков. В отличие от предшествующих работ, в которых на входной язык специализатора накладывались значительные ограничения, разработанный метод поддерживает все основные конструкции объектно-ориентированных языков, таких как C# и Java.

Существенной особенностью описываемого метода является использование нового понятия: *ВТ-кучи*. ВТ-куча является абстрактным описанием состояния реальной кучи (объектов и массивов, созданных в динамической памяти во время исполнения программы) и позволяет описать разделение данных на вычисляемые во время специализации и не вычисляемые. Это, в частности, дает возможность успешно специализировать программы, в которых библиотеки классов используются для определения понятий из некоторой предметной области и служат средством реализации проблемно-ориентированных языков. В результате специализации в остаточной программе вспомогательные объекты уже не создаются, а при необходимости вместо их полей используются локальные переменные.

Практическая значимость работы

Предложенный метод позволяет значительно повысить эффективность программ, в которых используются универсальные (но, возможно, неэффективные) библиотеки, вспомогательные проблемно-ориентированные языки и т.п. Это позволяет программисту не заниматься ручной оптимизацией таких программ, которая обычно требует много времени, приводит к трудно обнаруживаемым ошибкам, затрудняет последующее развитие и сопровождение программ. В результате повышается производительность труда программиста, повышается надежность программы, облегчается ее сопровождение.

Созданный метод может использоваться в компиляторах языков программирования и системах исполнения программ, позволяя без потери эффективности использовать методы, ускоряющие написание программ. Возможно использование разработанного метода в интегрированных средах разработки для анализа программ, поиска зависимостей, понимания чужого кода и т.д.

На основе предложенного метода разработан и реализован экспериментальный специализатор CILPE для промежуточного объектно-ориентированного языка CIL платформы Microsoft .NET.

Апробация работы и публикации

Результаты работы докладывались и обсуждались на:

- пятой международной конференции «Перспективы систем информатики» PSI'03, Россия, Новосибирск, Академгородок, 2003;
- конференции «Технологии Microsoft в научных исследованиях и высшем образовании», Россия, Москва, 2003;
- международной конференции «Microsoft Research Academic Conference: Compiler Architecture and Programming Languages», Венгрия, Будапешт, 2003;
- конференции «Microsoft Research Academic Days», Россия, Санкт-Петербург, 2004;
- Всероссийской конференции студентов, аспирантов и молодых ученых «Технологии Microsoft в теории и практике программирования», Россия, Москва, 2005;
- первом международном семинаре по метавычислениям Meta'08 «First International Workshop on Metacomputation in Russia» Meta'08, Россия, Переславль-Залесский, 2008;
- объединенном научном семинаре по робототехническим системам ИПМ им. М.В. Келдыша РАН, МГУ им. М.В. Ломоносова, МГТУ им. Н.Э. Баумана, ИНОТиИ РГГУ и семинаре отделения «Программирование» ИПМ им. М.В. Келдыша РАН, Россия, Москва, 2009;
- научном семинаре ИСП РАН, Россия, Москва, 2009;
- научном семинаре ЗАО «Авикомп Сервисез», Россия, Москва, 2009;
- научном семинаре ИПС им. А.К. Айламазяна РАН, Россия, Переславль-Залесский, 2009;
- одиннадцатой Всероссийской научной конференции «Научный сервис в сети Интернет: масштабируемость, параллельность, эффективность», Россия, Новороссийск, 2009.

По результатам работы имеются 7 публикаций, включая 1 статью в рецензируемом научном журнале из списка ВАК [6], 1 статью в международном периодическом издании [1], 1 статью в сборнике трудов международного научно-практического семинара [5], 4 статьи в сборниках трудов всероссийских конференций [2-4, 7].

Структура и объем диссертации

Диссертация состоит из введения, 5 глав, заключения, списка литературы и приложения. Содержание работы изложено на 183 страницах, из них 148

страниц основного текста и 27 страниц приложения. Список литературы включает 51 наименование. В работе содержится 193 рисунка и 6 таблиц.

Содержание работы

Во введении обоснована актуальность диссертационной работы, сформулированы цели и задачи, аргументирована научная новизна исследований, показана практическая значимость полученных результатов. Описана структура диссертации.

В первой главе описывается сущность метода частичных вычислений и возможности его использования для специализации программ. Дается обзор существующих подходов к частичным вычислениям для программ на объектно-ориентированных языках и формулируются минимальные требования, которым должны удовлетворять специализаторы, обрабатывающие программы на реальных объектно-ориентированных языках (таких как C# и Java).

В первом разделе главы 1 описывается классический подход к реализации частичных вычислений, часто применяемый в специализаторах программ, а также вводятся понятия и терминология, используемые в дальнейшем изложении.

Оптимизация программ на основе использования априорной информации о значениях части переменных, о контексте использования, называется *специализацией*.

Для примера рассмотрим программу $f(x,y)$ 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 одного аргумента $g(y)$, обладающая следующим свойством: $f(a,y)=g(y)$ для любого y. Как правило, программа g эффективнее программы f.

Одним из широко используемых методов специализации является метод частичных вычислений (Partial Evaluation, PE).

Метод частичных вычислений основан на разделении операций и других программных конструкций на *статические* (S) и *динамические* (D). В процессе частичных вычислений операции над статическими (известными) данными исполняются, а над динамическими (неизвестными) — переносятся в остаточную программу. Остаточная программа зависит только от неизвестной (на стадии специализации) части аргументов.

При этом понятие «статические операции и конструкции» не следует путать с понятием «статические методы и классы» (static), которое используется в объектно-ориентированных языках программирования, например, C# и Java.

Часть метода специализации, отвечающая за разделение операций и данных на статические и динамические, называется *анализом времен связывания* (Binding Time Analysis, BTA, BT-анализ) (рис. 1). Вторая часть метода

специализации, отвечающая за вычисление статической части программы и выделение динамической части в отдельную программу, называется *генератором остаточной программы* (Residual Program Generator, RPG). В этой части, собственно, и происходят частичные вычисления.

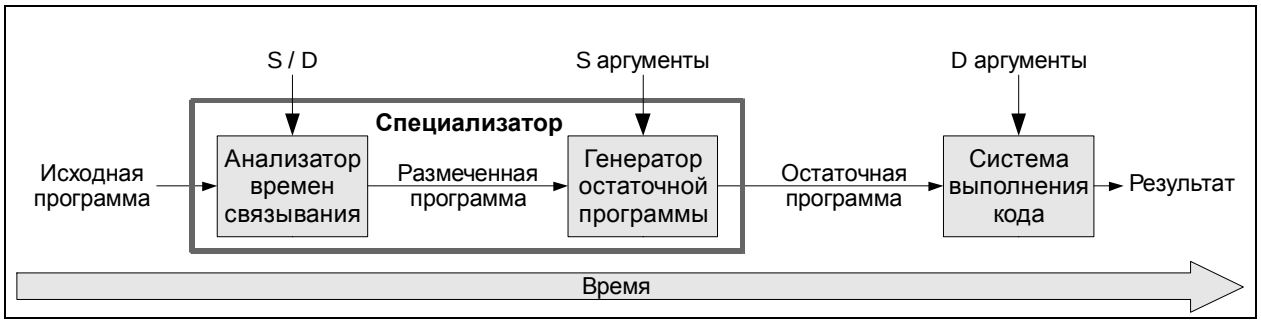

Рис. 1. Структура специализатора, основанного на методе частичных вычислений.

Для описания разделения на статические и динамические конструкции применяется разметка программы. Разметка S показывает, что конструкция статическая, D — динамическая.

Цель анализа времен связывания — по исходной программе и начальной разметке аргументов и опциональной разметке других частей программы построить некоторую корректную разметку всей программы, чтобы статические операции в таком разделении всегда получали только статические данные, а операции, которые в каких-то случаях могут получить динамические данные, были размечены как динамические.

Для заданной программы и начальной разметки, как правило, существует несколько корректных разметок. Чем больше операций будет отнесено к статическим, тем более эффективной будет остаточная программа.

Возможно несколько вариантов BT-анализа:

1. *Моновариантный по переменным* — каждую переменную разрешается разметить одним способом независимо от точки ее использования в программе.
2. *Поливариантный по переменным* — разрешается в разных частях программы размечать 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.
3. *Моновариантный по операциям* — запрещается преобразовывать и размножать код внутри метода. Следовательно, каждая операция размечается только одним способом независимо от пути вычисления, по которому можно прийти к данной операции.
4. *Поливариантный по операциям* — можно преобразовывать и размножать код внутри метода, что позволяет одному участку кода исходной программы сопоставить несколько участков выходной размеченной

программы с разными ВТ-разметками. Это позволяет более качественно разметить программу, отнеся к статическим заметно большую часть программы. В то же время существует опасность многократного роста размера программы.

5. *Моновариантный по методам* (подпрограммам) — каждый метод размечается только одним способом. В этом случае каждый метод будет размечен одним способом независимо от контекста его использования.
6. *Поливариантный по методам* (подпрограммам) — каждый метод может быть размечен в зависимости от его использования, в зависимости от разметки аргументов и результатов.
7. *Моновариантный по классам* — каждый класс размечается только одним способом. В таком варианте все объекты данного класса будут одинаково проспециализированы, независимо от особенностей их использования.
8. *Поливариантный по классам* — каждый класс может иметь несколько разметок. Это позволяет по-разному разметить различные использования объектов данного класса.

Как будет показано ниже, для успешного применения специализатор обязан быть поливариантным по всем характеристикам.

Вторая часть метода частичных вычислений — генерация остаточной программы. По корректно размеченной программе и значениям статических аргументов генератор остаточной программы строит окончательный результат специализации: остаточную программу.

Генератор остаточной программы производит обобщенное выполнение программы: статические инструкции выполняются, а динамические инструкции переходят в остаточную программу.

Во втором разделе главы 1 дается обзор существующих методов на базе метода частичных вычислений для объектно-ориентированных языков, указываются их ограничения. Формулируются требования к специализатору на основе метода частичных вычислений.

В университете города Токио под руководством Акинори Ёнэдзава (Akinori Yonezawa) и Хидэхико Масухара (Hidehiko Masuhara) разрабатывается специализатор для байт-кода языка Java, который можно было бы встроить в JIT-компилятор байт-кода. Описанный в работе 1999 года специализатор обрабатывает только примитивные данные и статические методы. Допускается различная разметка переменных в различных точках программы. В работе показано, как для метода построить систему ограничений на разметку локальных переменных и инструкций. В работе 2002 года группой Рейналд Аффелдт (Reynald Affeldt), Хидэхико Масухара (Hidehiko Masuhara), Эйдзиро Сумми (Eijiro Summi), Акинори Ёнэдзава (Akinori Yonezawa) предложено расширение специализатора, позволяющее обрабатывать и объекты. Разметка переменных

описывается деревьями: S или D в вершине дерева определяет, статический или динамический объект будет находиться в этой переменной, а поддеревья определяют разметку полей объекта.

BT-анализ метода основан на разделении объектов на локальные объекты (которые создаются внутри этого метода) и нелокальные (которые передаются методу извне). Локальные объекты делятся на те, которые могут быть выданы в качестве результата работы метода, и те, которые используются только внутри метода. Нелокальные объекты разделяются на те, в поля которых может осуществляться присваивание, и те, в поля которых присваивание не происходит. Инструкции создания локальных объектов, которые могут быть выданы из метода, и присваивания в поля нелокальных объектов обязаны размечаться D и переходить в остаточную программу. После разделения строится разметка всех инструкций в программе.

В обеих работах описаны специализаторы ограниченных подмножеств языка Java. Для разметок объектов используются деревья, что не позволяет эффективно размечать, выделяя необходимую статическую часть, графовые структуры из объектов. Методы являются моновариантными, что ограничивает область применимости методов.

В Королевском университете (Royal Veterinary and Agriculture University) в городе Копенгагене в Дании Питер Бертелсен (Peter Bertelsen) занимался семантикой языка Java и анализом времен связывания для Java-программ. В своей работе 1999 года П. Бертелсен рассматривает подмножество байт-кода языка Java. Это подмножество содержит инструкции для работы с локальными переменными, операции на стеке с примитивными данными и массивами и инструкции перехода Goto и If. Нет объектов и методов. Специализируемая программа — это одна последовательность инструкций, которая может содержать инструкции условного и безусловного переходов.

В работе описаны допустимые разметки и ограничения на корректную разметку программы и предложен метод нахождения решения для системы ограничений. Разработан генератор остаточной программы, который по размеченной программе и значениям статических аргументов строит остаточную программу.

В университете города Реннес (Rennes) во Франции Ульрик Шульц (Ulrik Pagh Schultz) в рамках своей диссертации разработал частичный вычислитель для подмножества байт-кода языка Java (1999-2004 года). Описанное подмножество относительно широко: можно описывать классы, конструкторы и методы. Разрешены операции над примитивными данными и виртуальные вызовы. Но отсутствуют операции работы с массивами, возможность изменять значения полей объектов, тело метода представляется не в виде последовательности инструкций, а в виде одного выражения.

Как и в предыдущих работах, программе сопоставляется система огра-

ничений на разметку. Но в отличие от них, разметка строится для описаний классов, а не для переменных в программе. Разметка программы находится путем решения системы ограничений. На основе размеченной программы генератором остаточной программы по известным значениям статических переменных строится остаточная программа.

Разработанный специализатор поддерживает ограниченное подмножество языка Java. Специализатор является моновариантным: каждая операция может иметь только одну разметку, каждый класс может иметь только одну разметку. Причем разметка класса и разметка классов его полей должны совпадать.

В следующих главах У. Шульц формулирует усовершенствования, которые необходимы для работы с реальными программами:

1. class polyvariance — класс размечается несколькими способами и каждой операции создания объекта или массива *new* соответствует одна из разметок;
2. method polyvariance — для каждого вызова метода метод размечается отдельно;
3. alias polyvariance — анализ, помечающий, где могут быть одинаковые данные, а где — разные. Желательно, чтобы и alias analysis был поливариантным (размножал классы и методы).

Но математического аппарата для реализации такого рода усовершенствований не предлагается.

	Инструкции передачи управления Goto и If	Работа с	
		массивами	объектами
Х. Масухара	да	нет	да
П. Бертелсен	да	да	нет
У. Шульц	нет	нет	да

Таб. 1. Классификация специализаторов на базе метода частичных вычислений по полноте реализации объектно-ориентированного языка.

	Поливариантность по			
	переменным	инструкциям	методам	классам / массивам
Х. Масухара	да	нет	нет	да
П. Бертелсен	да	нет	нет	да
У. Шульц	нет	нет	нет	нет

Таб. 2. Классификация специализаторов на базе метода частичных вычислений по поливариантности.

В рассмотренных выше работах предложены различные методы частичных вычислений для подмножеств объектно-ориентированного языка Java. При этом на каждое из этих подмножеств наложены сильные ограничения,

которые обычно не выполняются для реальных программ на языке Java (таб. 1).

Для практического применения к объектно-ориентированным языкам метод частичных вычислений должен:

1. Обрабатывать инструкции передачи управления Goto и If.
2. Поддерживать работу с объектами и массивами, допускать возможность изменения полей объектов и элементов массива.
3. Обладать единым анализом времен связывания (не проводить отдельный alias analysis).
4. Обладать поливариантностью по переменным (допускать несколько различных разметок одной и той же переменной в различных точках программы) (таб. 2).
5. Обладать поливариантностью по инструкциям (допускать несколько различных разметок частей метода в зависимости от разметки переменных и стека) (таб. 2).
6. Обладать поливариантностью по методам (допускать несколько различных разметок метода в зависимости от разметки аргументов и результатов метода) (таб. 2).
7. Обладать поливариантностью по классам и массивам (допускать несколько различных разметок однотипных переменных) (таб. 2).
8. Обладать расширенной разметкой объектов и массивов: одни поля статических объектов могут иметь статическую разметку, в то время как другие поля — динамическую.

Во второй главе описывается разработанный автором стековый объектно-ориентированный язык SOOL (Stack Object-Oriented Language), который является входным и выходным языком специализатора CILPE.

Для формального описания метода специализации необходимо иметь описание обрабатываемого языка: формальное описание синтаксиса и семантики. Использование промышленных языков для этой цели не подходит.

В синтаксисе промышленных языков много «синтаксического сахара» — конструкций, сокращающих и облегчающих написание программ. Но такие конструкции только удлиняют описание методов анализа и преобразования программ.

Семантика таких языков обычно описывается с точки зрения реализации интерпретатора или компилятора. А для описания метода необходимо компактное, но в тоже время полное, описание языка.

Поэтому на базе широко используемых языков CIL (Common Intermediate Language, внутренний язык платформы Microsoft .NET) и JBC (Java ByteCode, внутренний язык Java-платформы) разработан язык SOOL — стековый объектно-ориентированный язык. Он отличается от CIL и JBC более компактным синтаксисом (всего 19 инструкций) и математически строго опи-

санной семантикой.

В экспериментальном специализаторе CILPE реализованы компиляторы с языка CIL в язык SOOL и обратно.

В первом разделе главы 2 описывается синтаксис языка SOOL.

Язык SOOL — это объектно-ориентированный язык, основанный на классах. Программа на языке SOOL — это набор классов. Каждый класс описывается

- именем;
- списком имен классов, от которых данный класс наследуется;
- списком полей;
- списком методов.

Имена классов — это ссылочные типы. В языке SOOL также имеются встроенные типы INTEGER и FLOAT для данных примитивных типов, особые типы NULLTYPE и OBJECT и конструктор ссылочных типов массива type[]. NULLTYPE описывает тип особого значения NULL. OBJECT является общим предком всех ссылочных типов. Каждое поле задается именем и типом. Метод — именем, типами аргументов и результатов и телом метода. Тело метода — это список объявлений локальных переменных и список инструкций. Переменная описывается именем и типом. Требуется, чтобы все имена (классов, полей, методов, переменных) были уникальны, за исключением имен переопределенных методов.

Все методы в языке SOOL являются *виртуальными*: тело вызываемого метода определяется по типу первого аргумента. Для этого на определения методов наложены следующие ограничения:

1. Тип первого аргумента метода некоторого класса должен быть данным классом.
2. Если два метода имеют одинаковые имена, то типы всех аргументов, кроме первого, и типы всех результатов одного метода должны совпадать с типами аргументов и результатов другого метода.
3. Все классы, в которых описан данный метод, должны образовывать *иерархию*: должен существовать класс, в котором описан данный метод, а все другие классы, в которых также описан данный метод, должны прямо или опосредованно наследоваться от этого класса.

В отличие от языков CIL и JBC, в которых у метода не более одного возвращаемого значения, в языке SOOL метод может иметь сколько угодно возвращаемых значений.

В языке SOOL всего 19 инструкций:

- Leave, Goto n, Branch n
- DuplicateStackTop, RemoveStackTop, LoadConst const
- UnaryOp op, BinaryOp op (арифметические операции, сравнение)

- LoadVar var, StoreVar var
- NewObject class, LoadField fld, StoreField fld, CallMethod mthd, CastObject type
- NewArray type, LoadLength, LoadElement, StoreElement

В языке SOOL присутствуют основные понятия языков CIL и JBC, кроме исключений, структур и передачи параметров по указателю. Данные ограничения были введены ради компактности изложения. Они непринципиальны, поскольку метод может быть расширен на исключенные понятия.

Во втором разделе главы 2 описывается семантика языка SOOL: определены понятие состояния интерпретатора и правила выполнения шага вычисления.

Определение семантики языка SOOL дано в виде операционной семантики, то есть правила, описывающие изменения состояния. Используются правила как в стиле small step semantics, так и в стиле big step semantics.

Правила в стиле small step semantics используются для описания изменения состояния при выполнении одной инструкции (рис. 2).

$$\boxed{\text{context} \vdash_{\text{i}} \text{instr} : \text{state}_1 \rightarrow \text{state}_2}$$

Рис 2. Общий вид правила выполнения инструкции.

Правила в стиле big step semantics используются для описания результата вычисления программы и методов. Они связывают начальное и конечное состояния при вычислении программы или метода (рис. 3).

$$\boxed{\text{context} \vdash_{\text{i}} \text{method} : \text{state}_1 \Rightarrow \text{state}_2}$$

Рис 3. Общий вид правила выполнения метода

В описании используются следующие правила:

- правило, описывающее вычисление программы (big step semantics);
- правило, описывающее вычисление метода (big step semantics);
- правила, описывающие вычисление последовательности инструкций (начиная с некоторой инструкции некоторого метода до завершения метода) (big step semantics);
- правила, описывающие вычисление отдельных инструкций (small step semantics).

При описании семантики используются состояния, описывающие состояние интерпретатора (виртуальной машины) языка SOOL. Состояния бывают трех видов:

- последовательность значений — для описания результатов вычисления программы;
- стек значений и куча — для описания изменения кучи и результатов вычисления метода;
- стек значений, значения локальных переменных и куча — для описания

изменения стека, значений локальных переменных и кучи при вычислении инструкций.

В диссертации приведены все 22 правила — полное описание семантики языка. Пример правила приведен на рис. 4.

$$\frac{\mathrm{TypeOf}_{\mathrm{heap}}(\mathrm{oref}) <_{\mathrm{prog}} \mathrm{FieldClass}_{\mathrm{prog}}(\mathrm{fld}) \quad \mathrm{oref} \neq \mathrm{NULL} \quad (\_, \mathrm{obj}) = \mathrm{heap}(\mathrm{oref})}{\mathrm{prog} \vdash_{\mathrm{i}} \mathrm{LoadField\ fld} : (\mathrm{oref}::\mathrm{stack}, \mathrm{env}, \mathrm{heap}) \rightarrow (\mathrm{obj}(\mathrm{fld})::\mathrm{stack}, \mathrm{env}, \mathrm{heap})}$$

Рис 4. Правило для выполнения инструкции LoadField fld.

Построенные правила компактно описывают семантику языка SOOL.

Аналогичные правила используются для описания типизации программ, анализа времен связывания и генерации остаточной программы. Эти правила используются в доказательстве корректности специализатора.

В третьем разделе главы 2 описывается типизация программ на языке SOOL.

Внутренние языки CIL и JBC, а также внешние языки C# и Java являются статически типизированными языками. Программа считается типизируемой, если по программе (в которой заданы типы аргументов и результатов методов, типы локальных переменных и типы полей объектов) можно найти типы аргументов всех операций в программе и эти типы аргументов подходят для операций. Типизация используется для проверки корректности программы во время написания программы (а не во время исполнения). Например, типизация гарантирует, что во время исполнения не происходят обращения к полям или методам объектов, которые не обладают данными полями или методами.

Типизация позволяет, с одной стороны, более эффективно исполнять программу: не проверяя корректность операций во время исполнения. С другой стороны, позволяет обнаружить многие ошибки во время компиляции программы, а не во время ее исполнения.

Язык SOOL также статически типизирован. В диссертации проверка программы на типизируемость описана в виде правил (всего 22 правила — по одному на каждое правило выполнения программы). По программе и этим правилам порождается система уравнений на типы. Если эта система разрешима, то программа типизируема.

Описанная типизация аналогична существующей типизации программ на языках CIL и JBC. Для разрешения таких систем уравнений могут использоваться стандартные методы, аналогичные тем, которые используются для этих языков.

В этом же разделе, одновременно с описанием правил, приводится доказательство, что если программа типизируема, то проверки, описанные в правилах интерпретации, всегда будут выполнены. Например, в правиле LoadField fld (рис. 4) проверка типов всегда будет выполнена: будут обраще-

ния к полю только таких объектов, которые содержат данное поле. Доказательство проводится индукцией по числу шагов интерпретатора и путем сопоставления правил интерпретации и правил типизации.

Правила и доказательство теоремы подчеркивают близость разработанного языка SOOL и промышленных языков программирования.

В третьей главе описывается первая часть метода частичных вычислений для программ на объектно-ориентированном языке SOOL — анализ времен связывания. В главе описывается разметка программы, требования корректности разметки и способы нахождения разметки. Построенная разметка используется во второй части метода частичных вычислений для генерации остаточной программы.

В первом разделе главы 3 описывается разметка программ.

Разметка состоит из двух частей: разметки методов и BT-кучи. Понятие BT-кучи — это новый элемент метода специализации.

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.

BT-куча состоит из BT-объектов, которые состоят из:

- идентификатора BT-объекта;
- разметки BT-объекта: S или D;
- списка ссылочных типов: имен классов или массивных типов;
- списка полей всех классов, которые описаны в этом BT-объекте;
- специального поля ELEMENT, если BT-объекту приписан хотя бы один массивный тип.

Полям объектов и элементам массивов сопоставлено BT-значение:

- если поле имеет примитивный тип, то BT-вид S или D;
- если ссылочный тип — то некоторый BT-объект из BT-кучи.

Таким образом, BT-куча — это абстракция обычной кучи времени исполнения программы: вместо обычных значений в полях хранятся BT-значения.

Следует обратить внимание на ключевое отличие BT-объектов от обычных объектов и разметок, которые описываются в предшествующих работах: BT-объект может определяться несколькими классами. В BT-объекте присутствуют все поля этих классов. Это позволяет размечать статически поля объектов, точный тип которых не может быть определен во время анализа.

Отметим, что в BT-куче может быть несколько BT-объектов, с одним и

тем же классом. Это поливариантность анализа по классам.

При выполнении исходной программы в каждый момент времени можно построить отображение текущего состояния кучи в BT-кучу (рис. 5). Диагональной штриховкой отмечены те объекты в куче, которым соответствуют BT-объекты с BT-видом S. А серым — с BT-видом D.

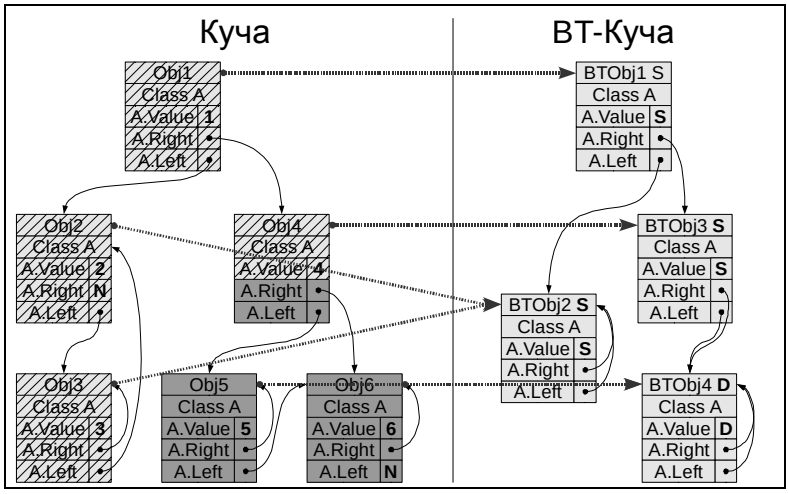

Рис. 5. BT-куча и связь с кучей времени исполнения.

Причем это отображение согласовано со ссылками между объектами и массивами, т.е. является гомоморфизмом. Пусть *F* — отображение, *obj* — объект с полем *fld* ссылочного типа, тогда *F(obj.fld) = F(obj).fld*.

Описанная BT-куча обладает следующими характеристиками, из-за которых разработанный метод частичных вычислений отличается от работ предшественников:

- она представляет собой граф;
- каждый BT-объект может описываться несколькими классами;
- каждый класс может встречаться в нескольких BT-объектах.

Именно эти особенности позволяют эффективно специализировать объектно-ориентированные программы.

Вторая часть разметки — это разметка методов. Она состоит из

- BT-объектов, приписанных каждому аргументу или результату метода, инструкции создания объектов или массивов BT-объекта. Они задают разметку аргумента или результата, локальной переменной, создаваемого объекта или массива соответственно;
- BT-вид S, D или X, приписанных каждой инструкции (BT-вид X используется только для инструкций). Они обозначают, что данная инструкция статическая, динамическая или особая соответственно. Этими BT-видами руководствуется генератор остаточной программы для построения остаточной программы.

В процессе анализа строятся разметки локальных переменных и элементов на стеке. Отображение между кучей и BT-кучей продолжается на отображение состояния в BT-состояние, которое включает в себя разметку элементов

стека, локальных переменных и BT-кучу.

Чтобы можно было построить данное отображение, разметка должна удовлетворять правилам согласованности.

Во втором разделе главы 3 описываются правила корректности разметки программы. Данные правила, как и правила типизации программы, описывают требования к разметке.

Чтобы проверить, что разметка удовлетворяет правилам корректности, необходимо в каждой точке программы построить BT-состояние: разметку элементов стека, локальных переменных и BT-кучу (которая входит в разметку программы), — и проверить, что все BT-состояния согласованы между собой согласно правилам.

Заметим, что одна и та же локальная переменная или один и тот же элемент стека в разных BT-состояниях могут иметь разные BT-разметки. Это — поливариантность по локальным переменным и элементам стека.

Каждое правило связывает BT-состояние до и после выполнения программы, выполнения метода или выполнения инструкций. Правила анализа времен связывания (25 правил) отображают семантику на понятия разметки и BT-кучи. Например, возможны три способа разметки инструкции LoadField fld в зависимости от разметки объекта и поля (рис. 6): инструкция размечается S, если объект и его поле размечены статически; D — если объект и поле размечены динамически; X — если объект размечен статически, а поле — динамически.

$$\frac{\begin{array}{c}(\_, \_, btObj) = btHeap(btOref) \qquad btVal = btObj(fld) \\ \text{либо } x = S,\ BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btVal) = S \\ \text{либо } x = X,\ BTKindOf_{btHeap}(btOref) = S,\ BTKindOf_{btHeap}(btVal) = D \\ \text{либо } x = D,\ BTKindOf_{btHeap}(btOref) = D,\ BTKindOf_{btHeap}(btVal) = D\end{array}}{prog,btHeap \vdash_{bt} LoadField^{x}\ fld : (btOref::btStack,btEnv) \rightarrow (btVal::btStack,btEnv)}$$

Рис 6. Правило BT-разметки инструкции LoadField fld.

Если разметка корректна, то можно построить в каждый момент выполнения программы отображение из состояния интерпретатора в BT-состояние. Такое отображение строится по очереди по шагам интерпретации.

Вначале строится отображение между начальными состояниями: аргументами программы и разметкой этих аргументов. Данное отображение задается пользователем как задание на построение разметки. Затем, выполняя правила для инструкций программы и правила анализа времен связывания, строятся отображения для каждого следующего шага:

- При создании нового объекта или массива данному объекту или массиву сопоставляется BT-объект, приписанный данной инструкции. Впоследствии это сопоставление никогда не изменяется.
- При выполнении инструкции перемещение данных из стека в локальные

переменные или кучу или назад происходит полностью аналогично перемещению разметок, поэтому все места (элементы стека, локальные переменные, поля объектов или элементы массивов), ссылающиеся на один и тот же объект, описываются одним и тем же ВТ-объектом.

В третьем разделе главы 3 приводится анализ методов решения систем правил.

Следует отметить, что построение разметки может быть связано с изменением исходной программы. Это позволяет поливариантно размечать методы и код методов.

Если не допускать изменения исходной программы, то для решения системы правил для заданной программы можно использовать известные методы. Система уравнений в данной ситуации конечна. Также конечно число инструкций создания объектов или массивов, поэтому конечно число ВТ-объектов и ВТ-куч. Следовательно, конечна область значений переменных в уравнениях. Для такой ситуации можно легко адаптировать известные методы. Например, метод constraint solving.

Но наибольшую эффективность метод частичных вычислений показывает, если он обладает поливариантностью по методам и коду методов. Т.е. обладает возможностью изменять программы в процессе анализа времен связывания.

Для нахождения разметки с одновременным изменением программы можно использовать модифицированный метод абстрактной интерпретации. Такой метод используется в реализованном экспериментальном специализаторе CILPE.

Чтобы нахождение разметки не происходило бесконечно долго, необходимы механизмы для завершения процесса абстрактной интерпретации и построения конечной размеченной программы. Для этого можно использовать различные эвристики. Самой простой является счетчик: не допускать размножения данного метода или кода метода более чем N раз (где N фиксировано). Как показано на примерах применения специализатора CILPE в приложении 1, даже использование такой эвристики позволяет получить желаемый эффект от специализации программ.

Следует отметить, что существует много, а за счет поливариантности иногда и бесконечно много, корректных разметок программы. В некоторых случаях из них нельзя выбрать наилучшую разметку. Поэтому для нахождения разметок можно использовать различные методы. Например, в одних ситуациях лучше использовать более быстрый метод (например, при специализации во время исполнения программы), а в других — более долгий (например, при специализации в интегрированной среде разработки, где время анализа менее критично и завершаемость процесса может контролировать программист). Однако подробный анализ достоинств и недостатков различных

методов поиска разметки заслуживает отдельного исследования и находится за рамками данной работы.

В четвертой главе описывается генератор остаточной программы для специализатора программ на стековом объектно-ориентированном языке SOOL.

Генератор остаточной программы — это второй и заключительный этап метода частичных вычислений. На вход генератору поступает размеченная программа и значения статических переменных. Генератор остаточной программы производит обобщенное выполнение программы:

- инструкции, размеченные S, выполняются;
- инструкции, размеченные D, переносятся в остаточную программу;
- инструкции, размеченные X, обрабатываются особо: частично выполняются, а в остаточную программу добавляются необходимые инструкции.

BT-вид X используется лишь для некоторых инструкций:

- создания объектов или массивов;
- доступа к локальным переменным, полям объектов или элементов массивов;
- вызова метода.

Описание работы генератора остаточной программы дано в виде правил выполнения (всего 21 правило). Правила показывают, как меняется состояние генератора остаточной программы и какие инструкции добавляются в остаточную программу при обработке одной размеченной инструкции входной программы. Пример правила приведен на рис. 7.

$$btProg \vdash_{rpg} LoadField^{X}\ fld : ((oref::stack, env, heap), ptr2var) \rightarrow (((stack, env, heap), ptr2var), [LoadVar\ ptr2var(oref, fld)])$$

Рис 7. Правило генерации остаточной программы для инструкции $LoadField^{X}$ fld.

В случае создания объекта или массива или доступа к полям или элементам такая разметка показывает, что происходит *обращение* к данным, которые частично статические, а частично — динамические.

Инструкции создания объектов или массивов выполняются: объекты или массивы создаются во время генерации остаточной программы. Но при этом в остаточной программе заводятся локальные переменные для динамических полей таких объектов или массивов. Инструкции доступа к полям объектов или элементам массивов превращаются в инструкции доступа к соответствующим локальным переменным (рис. 7).

Инструкции доступа к локальным переменным исходной размеченной программы заменяются на инструкции доступа к локальным переменным остаточной программы.

Инструкция вызова метода, в зависимости от указаний пользователя, либо заменяется инструкцией вызова специализированной версии метода, либо тело вызываемого метода подставляется вместо инструкции вызова.

По данным правилам построен генератор остаточной программы в специализаторе CILPE.

В пятой главе приведено доказательство корректности описанного метода специализации объектно-ориентированных программ.

Естественным и необходимым свойством любого специализатора программ является его *корректность*: при заданных ограничениях на условия работы программы, результат специализированной версии программы не должен отличаться от результата исходной программы.

Теорема (о корректности). Пусть даны размеченная программа f на языке SOOL и значения ее статических аргументов s. И пусть описанный генератор остаточной программы по f и s построил остаточную программу g. Тогда для любых значений динамических аргументов d программа $f(s,d)$ завершает работу тогда и только тогда, когда программа $g(d)$ завершает работу. Если обе программы завершают работу, то с одинаковым результатом.

Будем доказывать более сильное утверждение: что трасса вычислений $g(d)$ является подтрассой вычисления $f(s,d)$ (причем если трасса $f(s,d)$ бесконечна, то и трасса $g(d)$ тоже бесконечна). Доказательство проводится индукцией по числу шагов интерпретатора исходной программы.

Для доказательства будем проводить одновременную интерпретацию исходной размеченной программы, генерацию остаточной программы по исходной размеченной программе и интерпретацию остаточной программы.

С помощью разметки в каждой точке программы можно соединить состояния генератора остаточной программы (при обработке какой-то инструкции) и состояния интерпретатора остаточной программы (при обработке тех инструкций, которые были добавлены в программу на данном шаге генератора остаточной программы).

При генерации остаточной программы будут обработаны данные, которым соответствует BT-вид S. А все данные с BT-видом D, а также операции над ними, будут перенесены в остаточную программу. На рис. 5 слева показано состояние интерпретатора исходной программы. Диагональной штриховкой выделена куча при генерации остаточной программы, а серым — куча при интерпретации остаточной программы. В работе формально описано соединение состояний генератора остаточной программы и интерпретатора остаточной программы с помощью BT-состояния.

В доказательстве на каждом шаге сравниваются соединенное состояние и состояние интерпретатора исходной программы. В работе показано, что

1. (База индукции) В начальный момент времени состояние интерпретатора остаточной программы совпадает с соединенным состоянием.

2. (Шаг индукции) Если перед обработкой инструкции состояния совпадали, то и после выполнения инструкции они будут совпадать. Причем, если данная инструкция — инструкция условного перехода (или вызова виртуального метода), то интерпретатор исходной программы, генератор остаточной программы и интерпретатор остаточной программы перейдут на соответствующие друг другу ветви методов (или на соответствующие конкретные методы).

Показано, что состояния при любых значениях динамических аргументов всегда совпадают и управление передается соответствующим образом.

Это означает, что интерпретатор исходной программы дошел до конца программы тогда и только тогда, когда интерпретатор остаточной программы дошел до конца программы. Причем соединенное состояние генератора и интерпретатора остаточной программы совпадает с состоянием исходной программы.

Разметка результатов программы полностью динамическая (согласно правилу разметки программы). Значит, соединенное состояние совпадает с состоянием интерпретатора остаточной программы. То есть состояния интерпретаторов исходной и остаточной программ совпадают. Значит и результаты выполнения программ полностью совпадают. Теорема доказана.

В заключении сформулированы основные результаты работы.

В приложении 1 описан разработанный специализатор CILPE для промежуточного объектно-ориентированного языка CIL платформы Microsoft .NET. В специализаторе язык SOOL используется в качестве внутреннего языка, реализованы компиляторы с языка CIL в SOOL и обратно. Приведены примеры специализации объектно-ориентированных программ.

Основные результаты работы

- На основе проведенного исследования существующих методов частичных вычислений для функциональных и объектно-ориентированных программ разработан новый метод специализации программ на объектно-ориентированных языках, впервые обеспечивающий специализацию всех основных конструкций таких языков, как C# и Java, и обладающий поливариантностью. Повышение эффективности программ достигается за счет выполнения части операций во время специализации, удаления части объектов и изменения представления данных. Доказана корректность предложенного метода.
- Разработанный метод частичных вычислений реализован в экспериментальном специализаторе CILPE для промежуточного объектно-ориентированного языка CIL платформы Microsoft .NET и апробирован на модельных задачах. Показана возможность ускорения программ до 10 и более раз.

Список публикаций

1. A.M. Chepovskiy, And.V. Klimov, Ark.V. Klimov, Yu.A. Klimov, A.S. Mishchenko, S.A. Romanenko, S.Yu. Skorobogatov. Partial Evaluation for Common Intermediate Language // M. Broy and A.V. Zamulin (Eds.): Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers. — Lecture Notes in Computer Science, volume 2890/2003. — Springer-Verlag Berlin Heidelberg, 2003. — P. 171-177.
2. Климов Ю.А. Поливариантный анализ времен связывания в специализаторе CILPE для Common Intermediate Language платформы Microsoft.NET // Технологии Microsoft в теории и практике программирования: Труды Всероссийской конференции студентов, аспирантов и молодых ученых. Центральный регион. Москва, 17-18 февраля 2005 г. — М.: Изд-во МГТУ им. Н.Э. Баумана, 2005. — С. 128.
3. Климов Ю.А. О поливариантном анализе времен связывания в специализаторе объектно-ориентированного языка // Научный сервис в сети Интернет: технологии распределенных вычислений: Труды Всероссийской научной конференции (19-24 сентября 2005 г., г. Новороссийск). — М.: Изд-во МГУ, 2005. — С. 89-91.
4. Климов Ю.А. Генератор остаточной программы и корректность специализатора объектно-ориентированного языка // Научный сервис в сети Интернет: технологии параллельного программирования: Труды Всероссийской научной конференции (18-23 сентября 2006 г., г. Новороссийск). — М.: Изд-во МГУ, 2006. — С. 137-140.
5. Klimov Yu.A. An Approach to Polyvariant Binding Time Analysis for a Stack-Based Language // A.P. Nemytykh (Ed.): Proceedings of the First International Workshop on Metacomputation in Russia. Pereslavl-Zalessky, Russia, July 2-5, 2008. — Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008. — P. 78-84.
6. Климов Ю.А. Преобразование объектно-ориентированных программ в императивные методом частичных вычислений // Программные продукты и системы. — 2009. — № 2 (86). — С. 71-74.
7. Климов Ю.А. Метод частичных вычислений, позволяющий преобразовывать объектно-ориентированные программы в императивные // Научный сервис в сети Интернет: масштабируемость, параллельность, эффективность: Труды Всероссийской научной конференции (21-26 сентября 2009 г., г. Новороссийск). — М.: Изд-во МГУ, 2009. — С. 241-246.