# ОСОБЕННОСТИ ПОСТРОЕНИЯ СУПЕРКОМПИЛЯТОРА ЯЗЫКА JAVA

**Анд.В. Климов**

АННОТАЦИЯ.

Дается обзор особенностей методов суперкомпиляции объектно-ориентированных языков, реализованных в суперкомпиляторе языка Java. Первое отличие от суперкомпиляции функциональных языков: построение специализированного кода операций над объектами разделено на две стадии; сначала, в процессе собственно суперкомпиляции в остаточную программу переносятся (резидуализируются) все операции над объектами, затем избыточный код удаляется специально разработанным постпроцессингом. Второе: развертка дерева процессов и графа конфигураций производится вширь, а не вглубь, и на этой основе реализован ограниченный конфигурационный анализ управляющих конструкций языка Java. Третье: суперкомпилятор языка Java строится как система, управляемая человеком, а не как «черно-ящичная» автоматическая система. Обсуждаются мотивировки этих решений, состояние дел и планы на будущее.

## ВВЕДЕНИЕ

Построение надежных больших и сложных систем требует методов компонентного программирования и повторного использования универсальных компонентов, что приводит к потере эффективности программ во много раз. Для снятия такой неэффективности разрабатываются методы специализации программ, позволяющие переложить задачу порождения узкоспециализированного кода на компьютер. Среди методов *специализации* наиболее разработанными и перспективными для практики являются *частичные вычисления* [6, 12] и *суперкомпиляция* [15, 16, 17, 18, 1, 3, 11].

Изначально оба метода разрабатывались для функциональных языков. С небольшими доработками их можно применять для функциональных подмножеств императивных и объектно-ориентированных языков. Однако, этого недостаточно для удовлетворения естественному критерию полноты метода специализации: *при наличии полной информации о данных специализатор должен уметь выполнять все вычисления*. Кроме того, желательно извлекать как можно большую пользу из частично известной информации, порождая специализированный код.

В серии работ по *частичным вычислениям* для объектно-ориентированных языков ближе всего к достижению такой полноты находятся [2, 13, 8].

Автор *суперкомпиляции* В.Ф. Турчин разрабатывал ее применительно к языку Рефал [15, 16, 17, 18] в серии экспериментальных суперкомпиляторов. А.П. Немытых довел ее до конца [11] и вместе с коллегами вышел на красивые приложения [10]. Для широкого практического применения суперкомпиляция должна быть реализована для общепринятого в индустриальном программировании языка. Старые языки типа C++ для этого не годятся из-за их чрезмерной загроможденности машинно-ориентированными понятиями. К счастью, в последнее десятилетие в массовое программирование вышли языки «с управляемым кодом» — такие как Java и C#, имеющие достаточно строгую и прозрачную семантику.

Проект разработки Java-суперкомпилятора был начат вскоре после появления языка Java. Ниже дается общая характеристика изменений и дополнений к существующим методам суперкомпиляции для функциональных языков, которые потребовались для удовлетворения полноты специализации и были реализованы к настоящему времени в суперкомпиляторе JScp [5, 7].

## АВТОМАТИЧЕСКИЕ VS. УПРАВЛЯЕМЫЕ ЧЕЛОВЕКОМ ПРЕОБРАЗОВАНИЯ ПРОГРАММ

Методы специализации программ, в отличие от классических методов оптимизации, реализованных в промышленных компиляторах, вышли за пределы того, что можно использовать в «черно-ящичном», автоматическом режиме. Опыт их развития и тяжелых попыток применения привел к пониманию того грустного факта, что не получается встроить их внутрь систем так, чтобы пользователь не замечал их существования, — так же, как неприметно работают оптимизирующие преобразования в современных компиляторах. Чтобы быть незаметным для пользователя, метод должен работать примерно с линейной сложностью от размера кода. Однако, суперкомпиляция в общем случае имеет экспоненциальную, если не большую, сложность.

Универсальным методом борьбы с экспоненциальной сложностью является вовлечение человека в процесс вычислений. На повестке дня стоит задача создания удобных для пользователя понятий и средств, через которые работа специализаторов будет ему хорошо понятна и привязана к его текущим целям.

Важно отметить, что такая перспектива *не усложняет, а облегчает* разработку методов, имеющих много степеней свободы и требующих хитроумных стратегий принятия решений, если стараться построить автономный «черный ящик». Вместо изобретения стратегий лучше упрощать базовые алгоритмы и

подбирать разумные способы управления ими со стороны человека. В таком случае человек исполняет роль управляющей системы в рамках метасистемы (в смысле [14]).

Именно такой подход был принят при разработке суперкомпилятора языка Java. Чтобы подготовиться к решению задачи создания человеко-машинных систем преобразования программ, к настоящему моменту алгоритмы в Java-суперкомпиляторе управляются многими параметрами. Они задаются пользователем либо через опции командной строки (что достаточно для прогона простых примеров), либо из отдельного файла в специальном XML-формате, позволяющем задать параметры суперкомпиляции с точностью до метода и с учетом контекста его вызова. В настоящее время файл заполняется вручную с использованием в лучшем случае XML-редактора. В будущем предполагается разработать диалоговые средства в рамках какой-либо среды разработки Java-программ, например, Eclipse.

При разделении ролей между машиной и человеком машина «отвечает» за корректность преобразований программы, а человек — за достижение результата в разумное время. В идеале, никакие действия человека не должны «портить» программу. (Правда, велик соблазн время от времени отказываться от этого идеала, поскольку преобразования программ часто зависят от внешней информации, которую машина не может проверить.) Подобные системы А.П. Ершов называл «трансформационными машинами» [3]. Сейчас предстоит вернуться к этим идеям на новом уровне, имея за плечами опыт успехов и неудач построения практических специализаторов.

## ПЕРВАЯ ВЕХА ПРОЕКТА JSCP

Построение суперкомпилятора для языка с таким обилием понятий как Java должно быть пошаговым процессом. Нашей первой целью было: найти такой набор понятий и алгоритмов суперкомпиляции, которых будет достаточно для реализации *минимального суперкомпилятора, имеющего практический смысл*. После этого разработчики получают обратную связь от экспериментов по применению системы к реалистичным программам. Кроме того, только после этого можно начать разработку человеко-машинных интерфейсов для управления и понимания процессов суперкомпиляции. Сейчас проект JScp [7] находится примерно на этой стадии развития.

К настоящему времени метод суперкомпиляции достиг достаточно высокого уровня структуризации, чтобы при реализации конкретных суперкомпиляторов можно было выбирать из него различные подмножества средств.

Основное рабочее понятие суперкомпиляции — *конфигурация*. Это — обобщенное состояние процесса вычислений, представление которого похоже на представление состояния в интерпретаторе, но расширенное переменными, которые могут входить во все места, где может стоять обычное значение.

Метод суперкомпиляции состоит из двух основных частей:

- *прогонка (driving)* — построение *дерева всевозможных путей вычисления (процессов)* на некоторую глубину, начиная с некоторой конфигурации;
- *конфигурационный анализ* — операции и стратегии, обеспечивающие свертку потенциально бесконечного дерева процессов в конечный граф остаточной программы (т.н. *граф конфигураций*).

## ПРОГОНКА

Прогонка выполняет собственно частичные вычисления с учетом полноты информации, представленной в текущей конфигурации, и генерацию остаточного кода (*резидуализацию*) для всех примитивных операций языка, когда это нужно. От реализации прогонки в значительной степени зависит глубина и полнота специализации. Наш опыт показал, что прогонку нужно сразу реализовывать достаточно детально.

Однако, одновременно стало ясно, что от некоторых средств на первых этапах все-таки можно отказаться. А именно, в нынешней версии JScp *не* реализованы следующие средства, от которых не зависят основные алгоритмы прогонки и которые могут быть добавлены потом:

- *сужения* — распространение информации о проведенных проверках на равенство над значениями переменных с помощью применения подстановки к конфигурациям. (Оговоримся, что частный случай сужений после проверки на равенство целых чисел уже реализован). Традиционно сужения относятся к фундаментальным понятиям суперкомпиляции, но наш опыт показал, что они не так уж и необходимы;
- *дополнительные ограничения* на значения переменных: отношения «больше-меньше» и неравенство. Обычно эти средства рассматриваются как вторичные и реализуются в последнюю очередь. Так поступили и мы.

## КОНФИГУРАЦИОННЫЙ АНАЛИЗ

Традиционно конфигурационный анализ проводится разверткой дерева процессов в глубину (depth-first driving). При этом условие остановки (т.н. «свисток») в основном зависит лишь от одной ветви дерева

— пути в дереве процессов. Когда «свисток» укажет, что надо принудительно останавливаться, выполняется обобщение одной из конфигураций на данном пути.

Эта схема оказалась непригодной для языка Java, имеющего большой набор разнородных управляющих конструкций: условные **if** и **case**; циклы **for**, **while**, **do-while**; перехват исключений **try-catch**; синхронизированный блок **synchronized**. Последние две конструкции предполагают особый режим выполнения вложенных операций, а не просто задают разные схемы передач управления, как у первых двух конструкций. Такой разнобой делает весьма непростым определение единого процесса прогонки вглубь с учетом всевозможных комбинаций объемлющих блоков.

Поэтому в суперкомпиляторе JScp принято другое решение: развертка дерева процессов происходит вширь (width-first driving). Для каждой управляющей конструкции рекурсивно вызывается суперкомпиляция тела. Конец тела управляющей конструкции является естественным локальным ограничением глубины суперкомпиляции тела. Вместе с остаточным кодом тела выдаются конфигурации на всех выходах из тела. Остаточный код для всей конструкции строится после анализа всех выходных конфигураций, то есть сравнения их между собой и выполнения обобщения некоторых конфигураций.

В целом эта схема более похожа на анализ управляющих конструкций в классических оптимизирующих компиляторах, чем традиционный конфигурационный анализ для функциональных языков. Однако, конфигурационный анализ «вглубь» по В.Ф. Турчину более полон в том смысле, что в нем можно определить всегда завершающиеся автоматические стратегии. Наш нынешний конфигурационный анализ предполагает, что глубину раскрытия методов (inlining) определяет человек, пользуясь упомянутыми выше средствами задания параметров суперкомпиляции данной программы. Однако, методы не противоречат друг другу, и в будущем стратегии анализа «в глубину» могут быть добавлены к нашему анализу «вширь» «по конструкциям».

## ПРОГОНКА ОПЕРАЦИЙ НАД ОБЪЕКТАМИ

Понятие объекта, изменяющегося во времени вычисления и идентифицируемого уникальной ссылкой, — это главное отличие объектно-ориентированных языков от функциональных. Методы специализации функциональных языков опираются на следующее свойство значений: в любой момент времени специализации представление любого значения в конфигурации может быть перенесено на время выполнения остаточной программы. Для этого генерируется остаточный код, который при исполнении порождает равное значение. Здесь важно, что многократные исполнения этого кода или его копии порождают равные между собой значения.

Объекты не удовлетворяют этому свойству. Во-первых, они изменяются во времени и надо различать одинаковость содержимого объектов и равенство ссылок на них. Во-вторых, можно сгенерировать код, порождающий объект с таким же содержимым, но нельзя породить объект с «той же» ссылкой. Более того, даже при условии одинакового исходного состоянии программы, при многократных исполнениях кода выдаются разные объекты, с разными ссылками, хотя и с одинаковым содержимым.

Поэтому схема специализации операций над объектами отличается от прогонки операций над обычными значениями. Остаточный код операций над объектами порождается в две стадии:

1. При прогонке все операции над объектами резидуализируются, то есть строится их аналог в остаточной программе, даже если они полностью выполняются в период суперкомпиляции при известных данных. (Не требуется резидуализировать лишь операции чтения из полей объекта, и то при определенных условиях.) Это обеспечивает эквивалентность состояний объектов в суперкомпиляторе и в соответствующих точках остаточной программы при ее исполнении.
2. Избыточный код, порожденный таким способом, удаляется после завершения собственно суперкомпиляции, когда построена вся остаточная программа. Для этого реализован специальный постпроцессинг. Имея весь граф программы, можно выявить те переменные и операции, которые заведомо не нужны для вычисления результата.

Этот анализ реализует некоторое приближение: часть ненужных переменных или операторов может остаться. В нынешней версии анализ не учитывает порядка операций над объектами на графе программы и рассматривает их все как неупорядоченное множество операций. Такой алгоритм хорош тем, что он сходится быстрее, чем более точный, учитывающий структуру графа программы (скажем, методом абстрактной интерпретации). Тем не менее, он хорошо себя зарекомендовал на практике, вычищая подавляющую часть избыточного кода.

В постпроцессинге также реализованы упрощающие эквивалентные преобразования, свертки кода, делающие его более компактным и читабельным.

Возможен вопрос, требуется ли на самом деле реализация такого постпроцессинга? Ведь удаление избыточного кода — это типовая часть современных оптимизирующих компиляторов. Однако, дело в том, что подобные оптимизации — это всегда приближения, и по разным причинам (в частности, с учетом времени работы) их «затачивают» под определенные классы программ. Распространенные компиляторы

предполагают, что код либо написан человеком, либо порожден несложным препроцессором. Код, порожденный суперкомпилятором, сильно отличается от них.

Вторая причина в том, что нам важно было добиться как можно большей читабельности остаточной программы. Кроме банальной причины, что авторы Java-суперкомпилятора должны анализировать результаты и отлаживать его, остаточный код нельзя скрывать, исходя из перспективных целей создания человеко-машинных систем, в которых пользователю нужно видеть и понимать, что делает суперкомпилятор.

## ЭКСПЕРИМЕНТЫ

В настоящее время мы проводим эксперименты по специализации Java-программ разного стиля с помощью суперкомпилятора JScp. Анализ глазами остаточного кода показывает, что в целом глубина специализации превосходит наши ожидания в начале проекта. Особенно неожиданной оказалась достаточно высокая читабельность остаточной программы и понятность источников происхождения остаточных операторов из исходной программы. Видно, что при соответствующей поддержке в визуальной Java-студии можно будет сопоставлять остаточные и исходные операторы кликом мышки.

Среди небольших примеров, пожалуй, самый красивый — пример из статьи [19]. Это пример задачи из области численных вычислений. Ее автор показал, как, программируя в определенном стиле интерпретаторы выражений над сложными данными (числа, векторы, матрицы и т.п.), а затем специализируя программу интерпретатора при известном интерпретируемом выражении, можно получать нетривиальные оптимизации типа объединение («сплавление») циклов.

Стиль программирования интерпретатора из этого примера включал динамическое порождение выражений в виде совокупности объектов, задержку вычислений, продолжение вычислений задержанных выражений. Код интерпретатора использует такие высокоуровневые шаблоны объектно-ориентированного программирования как Visitor Pattern. В результате специализации неэффективные объектно-ориентированные шаблоны заменяются прямым кодом выражений на языке Java, да еще с проведением дополнительных оптимизаций. В результате наблюдается ускорение порядка десяти раз (в зависимости от характера выражений).

Автор статьи [19] показал лишь возможность таких преобразований с помощью реализованного им модельного частичного вычислителя для подмножеств языков C++ и Java. Суперкомпилятор JScp выдает аналогичный результат на основе общих методов для почти полного языка Java. Нет сомнения, что коллекция подобных приемов программирования с использованием специализаторов будет только расти.

Это открывает возможность создания объектно-ориентированных библиотек для дешевого компонентного программирования численных задач.

Из других потенциальных приложений JScp хочется отметить наши планы повторить на языке Java и Java-суперкомпиляторе многообещающие результаты [10] по *верификации* с помощью Рефал-суперкомпилятора [4] моделей протоколов, запрограммированных на Рефале. Для этого надо будет сделать некоторый шаг развития нашего ограниченного конфигурационного анализа в сторону полного анализа по В.Ф. Турчину.

## БЛАГОДАРНОСТИ

Автору доставило большую радость вести разработку и реализацию Java-суперкомпилятора совместно с В.Ф. Турчиным, Арк.В. Климовым и А.Б. Швориным. Мы благодарны нашим партнерам Ларри Витте, Бену Гертзелю и Юрию Мостовому за поддержку и консультирование по направлениям приложений суперкомпиляции для объектно-ориентированных языков. Сейчас работа выполняется при поддержке проектов РФФИ № 06-01-00574-а и № 08-07-00280-а и проекта Роснауки № 2007-4-1.4-18-02-064.

## ЛИТЕРАТУРА:

1. С.М. Абрамов "Метавычисления и их приложения". М.: Наука, 1995.
2. A.M. Chepovsky, 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, A.V. Zamulin, editors, Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers, volume 2890 of Lecture Notes in Computer Science, pages 171–177. Springer, 2003.
3. A.P. Ershov. "The Transformational Machine. Theme and Variations" // Proceedings on Mathematical Foundations of Computer Science 1981, August 31 – September 4, 1981, volume 118 of Lecture Notes in Computer Science, pages 16–32. Springer, 1981; А.П. Ершов "Трансформационная машина: тема и вариации" // Проблемы теоретического и системного программирования. Новосибирск, 1982. С. 5–24.
4. R. Glück, And.V. Klimov "Occam's razor in metacomputation: the notion of a perfect process tree" // P. Cousot, M. Falaschi, G. Filè, G. Rauzy, editors, Static Analysis Symposium. Proceedings, volume 724 of Lecture Notes in Computer Science, pages 112–123. Springer-Verlag, 1993.

5. B. Goertzel, And.V. Klimov, Ark.V. Klimov "Supercompiling Java Programs, white paper", 2002. http://www.supercompilers.com/white_paper.shtml.
6. N.D. Jones, Carsten K. Gomard, P. Sestoft "Partial Evaluation and Automatic Program Generation". Prentice-Hall, 1993.
7. Анд.В. Климов, Арк.В. Климов, А.Б. Шворин "Проект Java-суперкомпилятор". http://www.supercompilers.ru.
8. Ю.А. Климов "Особенности применения метода частичных вычислений к специализации программ на объектно-ориентированных языках". Препринт № 28, М.: ИПМ им. М.В. Келдыша РАН, 2008.
9. J.B. Kruskal "Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture" // Transactions of the American Mathematical Society, 95(2):210–225, 1960.
10. А.П. Лисица, А.П. Немытых "Верификация как параметрическое тестирование (эксперименты с суперкомпилятором SCP4)" // Программирование, 1:22–34, 2007.
11. А.П. Немытых "Суперкомпилятор SCP4: общая структура". М.: Наука, 2007.
12. S.A. Romanenko "A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure" // D. Bjørner, A.P. Ershov, N.D. Jones, editors, Partial Evaluation and Mixed Computation, pages 445–463. North-Holland, 1988.
13. U.P. Schultz, J.L. Lawall, C. Consel "Automatic program specialization for Java" // ACM Trans. Program. Lang. Syst., 25(4):452–499, 2003.
14. V.F. Turchin "The Phenomenon of Science". Columbia University Press, New York, 1977; В.Ф. Турчин "Феномен науки: Кибернетический подход к эволюции". Изд. 2-е. М.: ЭТС, 2000.
15. V.F. Turchin "The concept of a supercompiler" // Transactions on Programming Languages and Systems, 8(3):292–325, 1986.
16. V.F. Turchin "The algorithm of generalization in the supercompiler" // D. Bjørner, A.P. Ershov, N.D. Jones, editors, Partial Evaluation and Mixed Computation, pages 531–549. North-Holland, 1988.
17. V.F. Turchin "Metacomputation: Metasystem transitions plus supercompilation" // O. Danvy, R. Glück, P. Thiemann, editors, Dagstuhl Seminar on Partial Evaluation, volume 1110 of Lecture Notes in Computer Science, pages 481–509. Springer, 1996.
18. V.F. Turchin "Supercompilation: techniques and results" // D. Bjørner, M. Broy, I.V. Pottosin, editors, Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, June 25-28, 1996, Proceedings, volume 1181 of Lecture Notes in Computer Science, pages 227–248. Springer, 1996.
19. T. Veldhuizen. "Just when you thought your little language was safe: Expression Templates in Java" // G. Butler, S. Jarzabek, editors, Generative and Component-Based Software Engineering, Second International Symposium, GCSE 2000, Erfurt, Germany, October 9–12, 2000, Revised Papers, volume 2177 of Lecture Notes in Computer Science, pages 188–202. Springer, 2001.