МЕТОД ЧАСТИЧНЫХ ВЫЧИСЛЕНИЙ, ПОЗВОЛЯЮЩИЙ ПРЕОБРАЗОВЫВАТЬ ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ ПРОГРАММЫ В ИМПЕРАТИВНЫЕ

Ю.А. Климов

Введение

В настоящее время наиболее широкое распространение получили объектно-ориентированные языки программирования. В то же время для численных расчетов наиболее часто используются императивные языки программирования, например, C/C++ и Fortran.

Такие широко используемые объектно-ориентированные языки как C# или Java не получили широкого распространения в областях, связанных с высокой производительностью, потому что использование классов в некоторых случаях может заметно сказаться на производительности программ [17].

Для широкого использования объектно-ориентированных языков в численных расчетах необходимы средства преобразования программ, которые позволяют преобразовывать высокоуровневые объектно-ориентированные программы в низкоуровневые императивные.

Для таких преобразований можно использовать метод частичных вычислений [5,6], расширенный и адаптированный для объектно-ориентированных программ [8].

На основе метода частичных вычислений создан специализатор CILPE [3], способный выполнять описанные преобразования [9].

В данной работе описаны особенности ядра метода частичных вычислений — анализа времен связывания, которые позволяют справляться с поставленной задачей.

Метод частичных вычислений

Оптимизация программ на основе использования априорной информации о значении части переменных называется специализацией [5,6,16]. 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* является новая программа одного аргумента *g(y)*, обладающая следующим свойством: *f(a,y)=g(y)* для любого *y*.

Одним из широко используемых методов специализации является метод частичных вычислений (Partial Evaluation, PE) [5,6]. Данный метод заключается в получении более эффективного кода на основе использования априорной информации о части аргументов и однократного выполнения той части кода, которая зависит только от известной части аргументов (и не зависит от неизвестной части).

В процессе частичных вычислений операции над известными данными исполняются, а над неизвестными — переносятся в остаточную программу. Остаточная программа зависит только от неизвестной (на стадии специализации) части аргументов и будет исполняться только тогда, когда значения этих аргументов станут известны. Цель частичных вычислений — генерация остаточной программы.

Ядро метода частичных вычислений — разделение операций и других программных конструкций на статические (S) и динамические (D). (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.) Статические операции будут выполнены во время специализации программы, а динамические — перейдут в остаточную программу. S и D будем называть BT-метками (Binding Time метками, метками времени связывания).

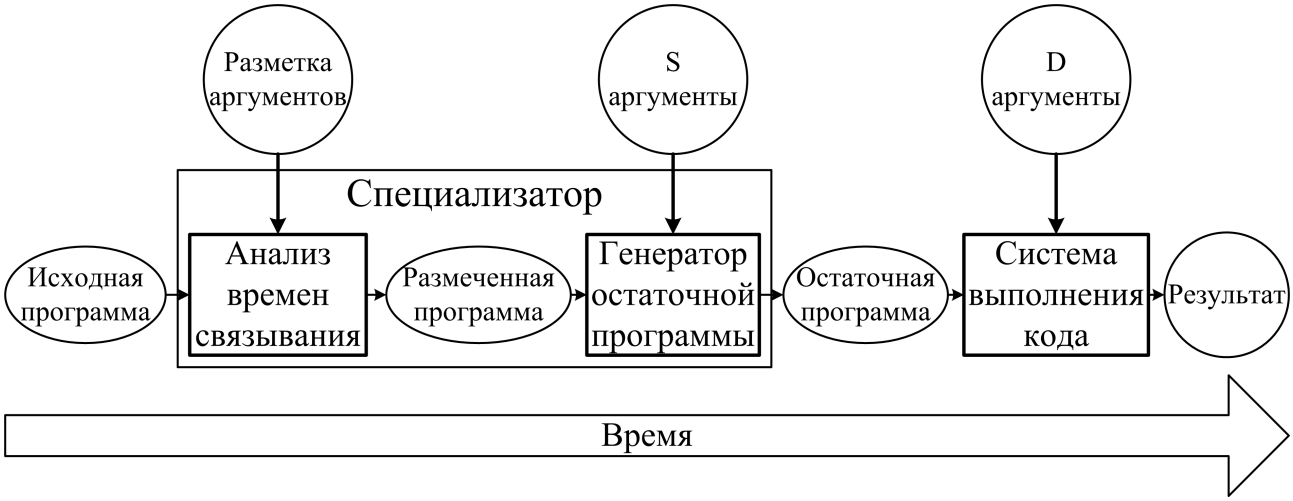

Рис. 1. Общепринятая структура специализатора, основанного на методе частичных вычислений.

Часть метода специализации, отвечающая за разделение операций и данных, называется анализом времен связывания (Binding Time Analysis, BTA, BT-анализ) (рис. 1) [7,11,12]. 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) [13]. В этой части, собственно, и происходят частичные вычисления.

Метод частичных вычислений хорошо развит для функциональных языков программирования [6]. Для объектно-ориентированных языков существуют специализаторы, основанные на методе частичных вычислений [1,2,14,15]. Но они, в отличие от специализатора CILPE, не решают в полной мере задачу по преобразованию объектно-ориентированных программ в императивные [8].

Анализ времен связывания для объектно-ориентированных языков

Наиболее сложной частью метода частичных вычислений является анализ времен связывания [7,11,12] — построение корректной разметки программы. Существует множество корректных разметок одной и той же программы. Но чем большую часть программы анализу времен связывания удалось пометить статически (S), сохранив при этом корректность разметки, тем больше вычислений выполнится во время специализации. И, соответственно, тем меньше вычислений перейдут в остаточную программу.

Основной проблемой при анализе объектно-ориентированных языков является наличие глобального изменяемого состояния, структурированного с помощью объектов, описанных классами. Во время анализа программы точный тип объектов не известен. Известен лишь надкласс объекта. Точный класс, являющийся наследником надкласса, будет известен только во время исполнения программы, когда все параметры заданы.

При использовании объектно-ориентированного языка стандартной является ситуация, когда описывается надкласс, лишь определяющий используемый интерфейс. При этом конкретные реализации описываются в подклассах. Во многих случаях не возможно с помощью анализа, не зная точных значений аргументов программы, вычислить точный тип объекта, который будет во время исполнения программы находиться в той или иной локальной переменной или поле объекта. Поэтому метод частичных вычислений должен обладать возможностью эффективно обрабатывать такие ситуации.

BT-куча для объектно-ориентированных языков

Для успешного решения выше поставленной проблемы, разметка объектно-ориентированных программ должна позволять преодолевать следующие трудности. Во-первых, значения полей объектов могут изменяться во время выполнения программы. Во-вторых, объекты могут ссылаться друг на друга, образовывая сложные графовые конструкции с циклами. И, в-третьих, точный тип объектов обычно не известен.

Для решения указанных проблем разработана разметка для объектно-ориентированных программ, состоящая из двух частей. Первая часть — программа, в методах которой инструкции помечены статически (S), динамически (D) или особой меткой (X), а полям объектов, аргументам и результатам методов, локальным переменным приписаны BT-значения (Binding Time значения, значения времени связывания) из второй части разметки [12].

Вторая часть разметки — BT-куча (Binding Time куча, куча времени связывания), определяющая разметку аргументов и результатов метода, локальных переменных и элементов стека.

BT-куча (аналогично куче во время исполнения программы) состоит из BT-объектов, связанных между собой ссылками. Каждый BT-объект состоит из трех частей: BT-метки S или D всего объекта, списка классов, списка полей с их BT-значениями. BT-значение — это либо ссылка на BT-объект, если соответствующее поле имеет объектный тип, либо BT-метка S или D, если поле имеет примитивный тип.

Принципиальное отличие от обычной кучи заключается в том, что BT-объект имеет список классов, а обычный объект — только один класс. Этот список показывает, какой конкретный тип может иметь объект во время исполнения программы. Например, если локальной переменной сопоставлен BT-объект с классами *B* и *C*, то во время исполнения в этой переменной может находиться объект класса *B* или класса *C* но не другого класса, например, *F* (рис. 2). BT-объект должен содержать все поля, которые определены во всех описанных в нем классах.

Рассмотрим для примера следующее описание классов на языке C#:

```
abstract class A {    int Fld1;    }
class B : A {    B Fld2;    }
class C : A {    A Fld3;    C Fld4;    }
```

Описаны абстрактный класс *A* и два его наследника *B* и *C*. Класс *A* имеет поле *Fld1* типа *int*, класс *B* — поле *Fld2* типа класс *B*, класс *C* — поля *Fld3* и *Fld4* типа класс *A* и класс *C*, соответственно. Тогда BT-куча может быть следующей (рис. 2).

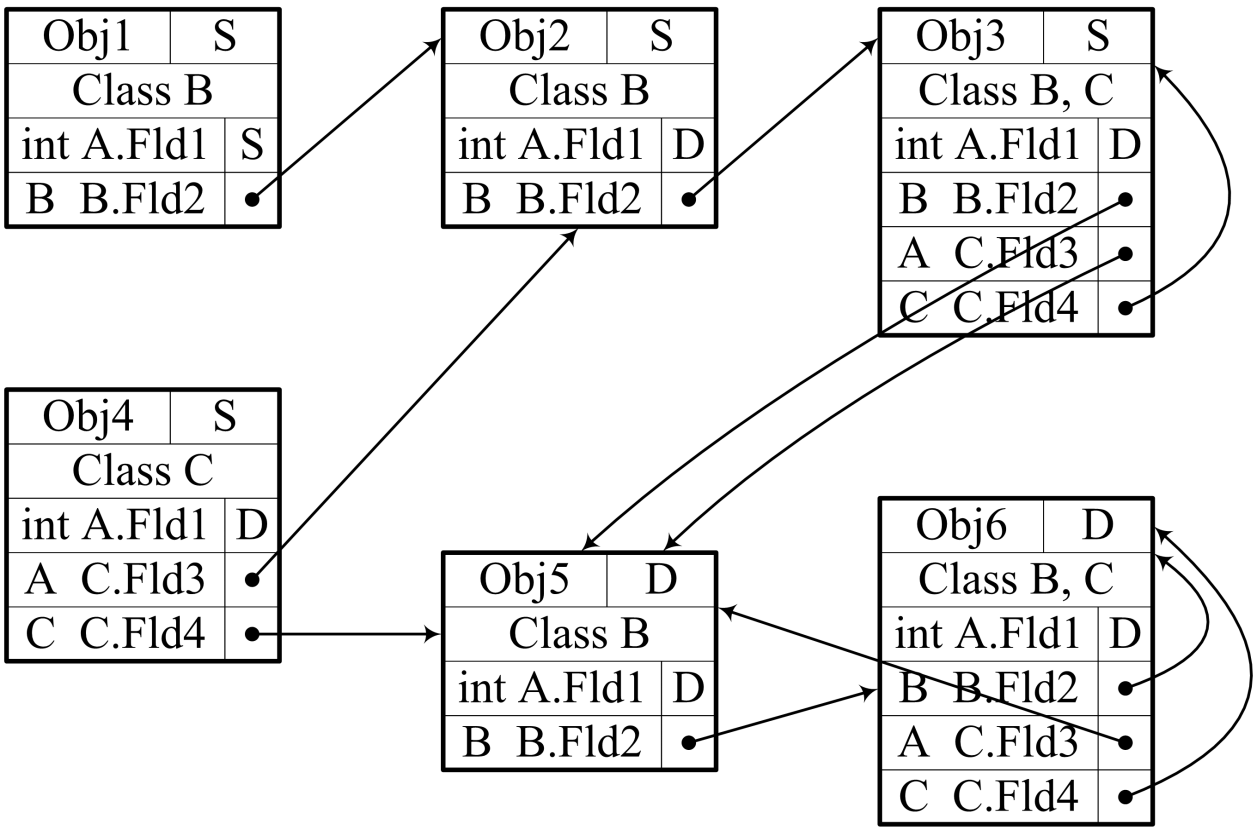

Рис. 2. BT-куча.

В описании BT-объекта Obj1 сказано, что поле Fld1 статическое, а разметка поля Fld2 — это BT-объект Obj2. Аналогично с BT-объектом Obj2. А BT-объект Obj3, на который ссылается поле Fld2 BT-объекта Obj2, описан двумя классами B и C. Данную разметку следует читать следующим образом. Если во время генерации остаточной программы в поле Fld2 объекта Obj2 находится объект класса B, то его поле Fld1 динамическое, а поле Fld2 описывается разметкой Obj5. А если во время генерации остаточной программы в поле Fld2 объекта Obj2 находится объект класса C, то его поле Fld1 также динамическое, а поля Fld3 и Fld4 имеют разметку Obj5 и Obj3 соответственно.

Таким образом, наличие нескольких классов в описании BT-объекта показывает, что анализ времен связывания не смог установить точный тип объекта. В то же время анализ времен связывания построил разметки полей на все возможные типы объектов. Это позволяет распространять информацию о статических полях даже в том случае, если точный тип не удалось установить.

BT-куча позволяет описывать любой сложности взаимные связи между объектами. Использование разработанной BT-кучи позволяет решить проблему неизвестности точного типа объекта на этапе анализа времен связывания.

Специализатор CILPE

Для формального описания метода частичных вычислений: разметки программ, анализа времен связывания и генератора остаточной программы, предложен модельный стековый объектно-ориентированный язык SOOL (Stack Object-Oriented Language) с формально описанными синтаксисом, семантикой и типизацией [10].

Для языка SOOL дано формальное описание BT-кучи и правил разметки программ [12]. Использование описанной разметки, однако, не сильно увеличивает сложность второй части метода частичных вычислений — генератора остаточной программы, который также формально описан [13].

Для апробации идей на основе формально описанного метода частичных вычислений создан специализатор CILPE [3] для внутреннего языка Common Intermediate Language (CIL) платформы Microsoft .NET [4].

Основным языком платформы Microsoft .NET является высокоуровневый объектно-ориентированный язык C#. Язык CIL хотя и не обладает всеми выразительными средствами языка C#, однако поддерживает все понятия языка C#. Это позволяет обрабатывать результат компиляции программ с языка C# на язык CIL без потери необходимой для специализатора информации о программе.

Специализатор CILPE поддерживает ограниченное, но широко используемое подмножество операций языка CIL. Не поддерживаются только исключения, передача аргументов по ссылке и структуры.

Специализатор CILPE разрабатывался с учетом особенностей объектно-ориентированных языков [8]. В результате специализатор CILPE способен выполнять операции над объектами или массивами, создание и использование которых прослеживается во время специализации программы [9]. Во многих случаях в результате специализации получается программа, в которой отсутствуют объекты, то есть происходит преобразование объектно-ориентированной программы в императивную программу.

Используемые в специализаторе CILPE техники и идеи могут быть легко адаптированы для других объектно-ориентированных языков, например, для Java байт-кода платформы Java.

Пример программы

Рассмотрим пример из работы [17], демонстрирующий «задержанные» вычисления над массивами. При реализации идеи «задержанных» вычислений порождается много вспомогательных объектов. Специализатор CILPE на основе метода частичных вычислений успешно выполняет все операции над такими объектами. В результате получаем программу в виде одного цикла без объектов.

Пример также демонстрирует, что при использовании специализатора нужно развивать новые нетривиальные приемы программирования, которые позволяют сочетать удобства компонентного программирования с высокой эффективностью результирующей программы.

Рассмотрим программу на языке C#, в которой поэлементно перемножаются два массива и к результату прибавляется третий массив:

```
// w = x + y * z
for (int i = 0; i < n; i++)
   w[i] = x[i] + y[i] * z[i];
```

При программировании на императивном языке явно указываются операции, которые необходимо сделать с каждым элементом массива. В реальных программах циклы по элементам массива могут занимать значительную часть программы.

Использование классов на объектно-ориентированным языке C# позволяет записать операции над массивами, не обращаясь явно к элементам массивов:.

```
// w = x + y * z
w.Assign(x.Plus(y.Times(z)));
```

Для описания операций в таком виде используем абстрактный класс Expr, описывающий интерфейс доступа к элементам массива (метод Get) и методы для поэлементного сложения и произведения массивов (методы Plus и Times):

```
abstract class Expr {
   public Expr () {}
   public abstract double Get (int i);
   public Expr Plus (Expr e) { return new BinaryOpExpr(this, e, new Plus()); }
   public Expr Times (Expr e) { return new BinaryOpExpr(this, e, new Times()); }
}
```

Отметим, что методы Plus и Times не вычисляют сумму массивов непосредственно в момент вызова. Они только запоминают необходимую операцию, «задерживая» вычисления, которая будет выполнена при вызове метода Get класса BinaryOpExpr.

Для представления массива и «задержанной» операции используем наследников класса Expr — класс Array и класс BinaryOpExpr:

```
class Array : Expr {
   double[] data;
   public Array (int n) { this.data = new double[n];  }
   public override double Get (int i) { return data[i]; }
   public void Assign (Expr e) { for (int i = 0; i < data.Length; i++) data[i] = e[i]; }
}
class BinaryOpExpr : Expr {
   Expr a, b; BinaryOp op;
   public BinaryOpExpr (Expr a, Expr b, BinaryOp op) { this.a = a; this.b = b; this.op = op; }
   public override double Get (int i) { return op.Apply(a.Get(i), b.Get(i)); }
}
```

Для присваивания значения элементам массива используется метод Assign, в котором описан цикл копирования значений из элементов аргумента в элементы массива.

Для описания операций над элементами массивов используется абстрактный класс BinaryOp. В данном классе определен единственный метод Apply — операция над двумя числами. А сами операции описываются классами Plus и Times — сложения и умножения соответственно:

```
abstract class BinaryOp {    public abstract double Apply (double x, double y);    }
class Plus : BinaryOp {    public abstract double Apply (double x, double y) { return x + y; }    }
class Times : BinaryOp {    public abstract double Apply (double x, double y) { return x * y; }    }
```

Специализация примера

Рассмотрим выше описанный пример, где w, x, y, z — это объекты класса Array, объявленные выше по программе:

```
// w = x + y * z
w.Assign(x.Plus(y.Times(z)));
```

Методы Plus и Times не производят вычислений. Они порождают объекты, описывающие необходимые операции с элементами массивов. Затем в методе Assign созданные объекты используются. И никуда более они не передаются.

В реальных вычислительных программах таких промежуточных объектов может быть очень много. Создание и обработка таких объектов может заметно сказаться на производительности программы. Поэтому нужен механизм, позволяющий преобразовывать такие программы в программы без создания промежуточных объектов.

Таким механизмом является специализатор CILPE, использующий метод частичных вычислений.

При специализации программы указанный выше участок кода преобразуется специализатором CILPE в цикл на стековом языке CIL. Для читаемости приведем его эквивалент на языке C#:

```
// w = x + y * z
for (int i = 0; i < n; i++)
   w[i] = x[i] + y[i] * z[i];
```

В результате специализации переменные w, x, y, z не объекты класса Array, а массивы языка C#.

Специализатор CILPE полностью выполнил операции над объектами классов BinaryOp, Plus, Times, Expr, Array и BinaryOnExpr, и в остаточной программе от них не осталось и следа. Таким образом, в результате специализации получилась императивная программа.

Заключение

Использование описанного в работе анализа времен связывания, позволило реализовать эффективный специализатор CILPE на основе частичных вычислений, который позволяет эффективно программировать вычислительные задачи.

При использовании объектов программа получается более короткой и выразительной. Что сокращает время разработки программы и упрощает отладку программы. В то же время после специализации исходной программы получается программа, по эффективности сравнимая с императивной программой.

По указанным свойствам специализатор CILPE превосходит известные нам существующие специализаторы для объектно-ориентированных языков [8].

Работа поддержана проектами РФФИ № 08-07-00280-а и № 09-01-00834-а.

ЛИТЕРАТУРА:

1. R. Affeldt, H. Masuhara, E. Sumii, A. Yonezawa "Supporting objects in run-time bytecode specialization" // In Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation, pp.50-60. ACM Press, 2002
2. P. Bertelsen "Binding-time analysis for a JVM core language" // April 1999, URL: ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Bertelsen/bta-core-jvm.ps.gz (дата обращения: 01.06.2009)
3. A.M. Chepovsky, An.V. Klimov, Ar.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.): PSI 2003, LNCS 2890/2003, pp. 171-177. Springer-Verlag Berlin Heidelberg, 2003. URL: http://www.springerlink.com/content/r4rr30ajjn449q1a/ (дата обращения: 01.06.2009)
4. Common Language Infrastructure (CLI) // URL: http://www.ecma-international.org/publications/standards/Ecma-335.htm (дата обращения: 01.06.2009)
5. А.П. Ершов "О сущности трансляции" // Программирование №5. Москва. 1977. С. 21-39
6. N.D. Jones, C.K. Gomard, P. Sestoft "Partial Evaluation and Automatic Compiler Generation" // C.A.R. Hoare Series, Prentice-Hall, 1993
7. Ю.А. Климов "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
8. Ю.А. Климов "Особенности применения метода частичных вычислений к специализации программ на объектно-ориентированных языках" // Препринты ИПМ им.М.В.Келдыша. 2008. № 12. 27 с. URL: http://library.keldysh.ru/preprint.asp?id=2008-12 (дата обращения: 01.06.2009)
9. Ю.А. Климов "Возможности специализатора CILPE и примеры его применения к программам на объектно-ориентированных языках" // Препринты ИПМ им.М.В.Келдыша. 2008. № 30. 28 с. URL: http://library.keldysh.ru/preprint.asp?id=2008-30 (дата обращения: 01.06.2009)
10. Ю.А. Климов "SOOL: объектно-ориентированный стековый язык для формального описания и реализации методов специализации программ" // Препринты ИПМ им.М.В.Келдыша. 2008. № 44. 32 с. URL: http://library.keldysh.ru/preprint.asp?id=2008-44 (дата обращения: 01.06.2009)

11. Yu.A. Klimov "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. Pp. 78-84. URL: http://meta2008.pereslavl.ru/accepted-papers/paper-info-6.html (дата обращения: 01.06.2009)
12. Ю.А. Климов "Специализатор CILPE: анализ времен связывания" // Препринты ИПМ им.М.В.Келдыша. 2009. № 7. 28 с. URL: http://library.keldysh.ru/preprint.asp?id=2009-7 (дата обращения: 01.06.2009)
13. Ю.А. Климов "Специализатор CILPE: генерация остаточной программы" // Препринты ИПМ им.М.В.Келдыша. 2009. № 8. 26 с. URL: http://library.keldysh.ru/preprint.asp?id=2009-8 (дата обращения: 01.06.2009)
14. U.P. Schultz "Object-Oriented Software Engineering Using Partial Evaluation" // PhD thesis, University of Rennes I, Rennes, France, December 2000
15. U.P. Schultz, J.L. Lawall, C. Consel "Automatic program specialization for Java" // ACM Transactions on Programming Languages and Systems 25 (4) (2003) 452-499
16. V.F. Turchin "The concept of a supercompiler" // ACM Transactions on Programming Languages and Systems, 8, 1986, pp.292-325
17. T.L. Veldhuizen "Just When You Thought Your Little Language Was Safe: "Expression Templates" in Java" // G. Butler and S. Jarzabek (Eds.) : GCSE 2000, LNCS 2177/2001, pp. 188-200. Springer-Verlag Berlin Heidelberg, 2001. URL: http://www.springerlink.com/content/p152067k1213k606/ (дата обращения: 01.06.2009)