Закрытое акционерное общество Научно-исследовательский институт
«Центрпрограммсистем»

Программные продукты и системы

НАУЧНОЕ И НАУЧНО-ПРАКТИЧЕСКОЕ ИЗДАНИЕ

№ 2 (86), 2009

Главный редактор

С.В. ЕМЕЛЬЯНОВ, *академик РАН*

Тверь

© ПРОГРАММНЫЕ ПРОДУКТЫ И СИСТЕМЫ

Международное научно-практическое приложение к международному журналу
“ПРОБЛЕМЫ ТЕОРИИ И ПРАКТИКИ УПРАВЛЕНИЯ”

SOFTWARE & SYSTEMS

Издается МНИИПУ, Главной редакцией международного журнала и НИИ “Центрпрограммсистем”

Журнал зарегистрирован в Комитете Российской Федерации по печати 26 июня 1995 г.

Регистрационное свидетельство № 013831

Подписной индекс в каталоге Агентства “Роспечать”
70799

ISSN 0236-235X

Главный редактор
С.В. ЕМЕЛЬЯНОВ, *академик РАН*

Научные редакторы номера:
С.М. АБРАМОВ, *доктор физико-математических наук, член-корреспондент РАН*
Н.А. СЕМЕНОВ, *доктор технических наук*

МЕЖДУНАРОДНАЯ РЕДАКЦИОННАЯ КОЛЛЕГИЯ

Боянов Борис – академик Болгарской Академии наук (Болгария, г. София)
Вагин В.Н. – д.т.н., профессор Московского энергетического института (Технического университета) (г. Москва)
Валькман Ю.Р. – д.т.н., Институт кибернетики им. В.М. Глушкова НАН Украины (Украина, г. Киев)
Голенков В.В. – д.т.н., профессор Беларусского государственного университета (Беларусь, г. Минск)
Еремеев А.П. – д.т.н., профессор Московского энергетического института (Технического университета) (г. Москва)
Куприянов В.П. – д.э.н., профессор, генеральный директор НИИ «Центрпрограммсистем», первый заместитель главного редактора международного журнала «Программные продукты и системы» (г. Тверь)
Курейчик В.М. – д.т.н., профессор Южного федерального университета (г. Таганрог)
Лисецкий Ю.М. – к.т.н., генеральный директор фирмы S&T (Украина, г. Киев)
Нгуен Тхань Нги – д.ф.-м.н., профессор, проректор Ханойского открытого университета (Вьетнам, г. Ханой)
Осипов Г.С. – д.ф.-м.н., профессор, НИИ информационных систем (г. Москва)
Палюх Б.В. – д.т.н., профессор Тверского государственного технического университета (г. Тверь)
Попков В.К. – д.ф.-м.н., профессор, академик МАИ (г. Новосибирск)
Поспелов Д.А. – д.т.н., профессор, академик РАЕН (г. Москва)
Решетников В.Н. – д.ф.-м.н., профессор, заместитель главного редактора международного журнала «Программные продукты и системы» (г. Москва)
Семенов Н.А. – д.т.н., профессор, заместитель главного редактора международного журнала «Программные продукты и системы» (г. Тверь)
Сотников А.Н. – д.ф.-м.н., профессор, Международный суперкомпьютерный центр РАН (г. Москва)
Сулейманов Д.Ш. – д.т.н., академик АН Республики Татарстан, профессор Казанского государственного технического университета (Россия, Республика Татарстан, г. Казань)
Тарасов В.Б. – к.т.н., Московский государственный технический университет им. Н.Э. Баумана (г. Москва)
Федотов Александр – д.филол.н., профессор Софийского университета им. св. Климента Охридского (Болгария, г. София)
Шейнман Арнольд – к.т.н., корпорация «Motorola» (США, г. Чикаго)
Язенин А.В. – д.ф.-м.н., профессор Тверского государственного университета (г. Тверь)

АДРЕС РЕДАКЦИИ

Россия, 170024, г. Тверь, пр. 50 лет Октября, 3а

Телефон (482-2) 39-91-49

Факс (482-2) 39-91-00

E-mail: RED@CPS.TVER.RU
WWW.SWSYS.RU

Подписано в печать 29.05.2009 г.
Отпечатано в типографии НТП “Фактор”
Россия, 170000, г. Тверь, а/я 0605
Заказ № 59
Выпускается один раз в квартал
Год издания двадцать второй
Формат 60×84 1/8
Тираж 1000 экз.
Объем 208 стр.
Цена 120 руб.

Авторские гонорары не выплачиваются
При перепечатке материалов ссылка на журнал обязательна

знакомых слов // Диалог'98: тр. Междунар. сем. по компьютерной лингвистике и ее приложениям. Казань: ООО «Хэтер», 1998. Т. 2. С. 547–552.

3. Сокирко А.В. Морфологические модули на сайте www.aot.ru // Диалог'2004: тр. Междунар. конф. М.: Наука, 2004. С. 559–564.

4. Ермаков А.Е., Плешко В.В. Компьютерная морфология в контексте анализа связного текста // Там же. С. 185–190.

5. Куршев Е.П., Кормалев Д.А., Сулейманова Е.А., Трофимов И.В. Исследование методов извлечения информации из текстов с использованием автоматического обучения и реализация исследовательского прототипа системы извлечения информации // Матем. методы распознавания образов: сб. докл. 13-й Всерос. конф. М.: МАКС Пресс, 2007. С. 602–605.

ПРЕОБРАЗОВАНИЕ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ В ИМПЕРАТИВНЫЕ МЕТОДОМ ЧАСТИЧНЫХ ВЫЧИСЛЕНИЙ

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

Ю.А. Климов (ИПМ им. М.В. Келдыша РАН, г. Москва, yuklimov@keldysh.ru)

Объектно-ориентированные языки обладают большей выразительностью, чем императивные. Но в некоторых случаях эффективность программ на императивных языках программирования заметно выше, чем на объектно-ориентированных. Как показано в статье, частичный вычислитель CILPE способен преобразовывать объектно-ориентированные программы в более эффективные императивные.

Ключевые слова: *специализация, частичные вычисления, объектно-ориентированное программирование, C#, CILPE.*

Для самых разных задач часто используемые модули можно оформить в отдельные библиотеки. Повторное использование таких библиотек заметно повышает как эффективность программирования, так и последующее сопровождение исходного кода. Отдельные модули можно отлаживать и оптимизировать независимо от кода основной программы, что повышает надежность и производительность программ.

В настоящее время наиболее широко распространены объектно-ориентированные языки программирования. Поэтому библиотеки стандартных подпрограмм часто оформляются в виде классов.

В некоторых случаях использование классов может заметно сказаться на производительности программ. Например, в случае численных расчетов затраты на обработку объектов оказываются достаточно большими [1].

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* [2], адаптированный для объектно-ориентированных программ [3].

На основе метода частичных вычислений создан специализатор *CILPE* [4], способный выполнять описанные преобразования [5].

Частичные вычисления

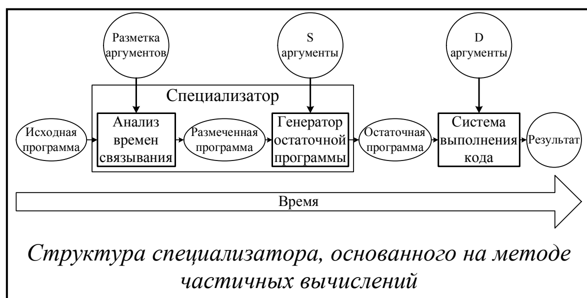
Оптимизация программ на основе использования априорной информации о значении части переменных называется *специализацией*. Рассмотрим программу **f(x, y)** от двух аргументов **x** и **y** и значение одного из ее аргументов **x=a**. Результатом специализации программы **f(x, y)** по известному аргументу **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 ч*астичных вычислений* (*Partial Evaluation*, *PE*) [2]. 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.

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.

Метод частичных вычислений основан на разделении 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.

Часть метода специализации, отвечающая за разделение операций и данных, называется *анализом времен связывания* (*Binding Time Analysis*, *BTA*, *BT*-анализ) (см. рис.). Вторая часть метода специализации, отвечающая за вычисление статической части программы и выделение динамической части в отдельную программу, называется



Структура специализатора, основанного на методе частичных вычислений

генератором остаточной программы (*Residual Program Generator*, *RPG*). В этой части, собственно, и происходят *частичные вычисления*.

Метод частичных вычислений хорошо развит для функциональных языков [2]. Для объектно-ориентированных языков существуют специализаторы, основанные на методе частичных вычислений [3]. Но они, в отличие от специализатора *CILPE*, не решают в полной мере задачу по преобразованию объектно-ориентированных программ в императивные.

Специализатор CILPE

На основе метода частичных вычислений создан специализатор *CILPE* [4] для внутреннего языка *Common Intermediate Language* (*CIL*) платформы *Microsoft.NET*.

Язык *CIL* является объектно-ориентированным стековым языком. Он не предназначен для ручного написания программ. Но все языки платформы *Microsoft.NET*, такие как *C*#, *J#*, *SML.NET*, компилируются в язык *CIL*. Наличие единого внутреннего языка позволяет реализовать специализатор только для этого языка, а затем использовать его для специализации программ, написанных на различных языках программирования.

Основным языком платформы *Microsoft.NET* является высокоуровневый объектно-ориентированный язык *C#*. Язык *CIL*, хотя и не обладает всеми выразительными средствами языка *C#*, поддерживает все его понятия. 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 [3]. В результате специализатор *CILPE* способен выполнить операции над объектами или массивами, создание и использование которых прослеживается во время специализации программы [5]. Во многих случаях в результате специализации получается программа, в которой отсутствуют объекты, то есть происходит преобразование объектно-ориентированной программы в императивную.

Используемые в специализаторе *CILPE* техники и идеи могут быть легко адаптированы для других объектно-ориентированных языков, например, для языка *Java Byte Code* (*JBC*) платформы *Java*.

Пример исходной программы

Рассмотрим пример из работы [1], демонстрирующий *задержанные* вычисления над массивами. При реализации идеи задержанных вычислений порождается много вспомогательных объектов. Специализатор *CILPE* на основе 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:

```
…
// 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 выполнить 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**. А также методы **Plus** и **Times** для поэлементного сложения и произведения массивов:

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

Атрибут метода **[Inline]** в описании классов показывает специализатору *CILPE*, что данный

метод должен быть раскрыт. То есть вместо вызова метода необходимо подставить его тело.

Для представления массива языка *C#* в виде наследника класса **Expr** определим производный класс **Array**. Его конструктор описывает создание массива, а метод **Get** обращение к элементам массива:

```
class Array : Expr {
  double[] data;
  [Inline] public Array (int n)
  { this.data = new double[n]; }
  [Inline] public override double Get (int i)
  { return data[i]; }
  private static int D (int n) { return n; }
  [Inline] public void Assign (Expr e) {
    for (int i = D(0); i < data.Length; i++)
      data[i] = e[i];
  }
}
```

Для присваивания значения элементам массива используется метод **Assign**. В нем описан цикл копирования значений из элементов аргумента в элементы массива. Отметим, что это единственный метод, в котором описан такого рода цикл. В дальнейшем при использовании классов **Expr** и **Array** такие циклы описывать не приходится.

Следует обратить внимание на методы **Plus** и **Times** класса **Expr**. Они предназначены для поэлементного сложения и умножения массивов. Но вместо явного вычисления, что потребовало бы заведения дополнительного массива для промежуточного результата, создается объект класса **BinaryOpExpr**, описывающий необходимые действия над элементами массива. То есть вместо того чтобы в этом методе непосредственно вычислять массив, операция над массивами запоминается и будет выполнена при обращении к элементам массива-результата.

Для этого используется класс **BinaryOpExpr**. Он хранит в себе два объекта класса **Expr** и операцию, которую необходимо произвести с элементами этих массивов, – элемент класса **BinaryOp**:

```
class BinaryOpExpr : Expr {
  Expr a, b; BinaryOp op;
  [Inline] public BinaryOpExpr (Expr a, Expr b,
    BinaryOp op) {this.a = a; this.b = b; this.op = op;}
  [Inline] public override double Get (int i)
  { return op.Apply(a.Get(i), b.Get(i)); }
}
```

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

Специализация примера

Допустим, что в теле программы используются описанные классы для вычисления поэлементного произведения двух массивов и суммирование с третьим; **w**, **x**, **y**, **z** – объекты класса **Array**, объявленные по исходной программе. Часть исходной программы на языке *C*#:

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

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

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

Таким механизмом является специализатор *CILPE*, использующий метод частичных вычислений.

При специализации программы указанный ранее 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** – уже 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.

Вышеописанный пример показывает, что использование объектно-ориентированного языка и специализатора, основанного на частичных вычислениях, например, языка *C*# и специализатора *CILPE*, позволяет эффективно программировать вычислительные задачи.

При использовании объектов программа получается более короткой и выразительной, что сокращает время ее разработки и упрощает отладку.

В то же время после специализации исходной программы получается программа, по эффективности сравнимая с императивной.

Для достижения уровня специализации, требуемого в таких задачах, разработанный специализатор *CILPE* обладает поливариантностью по коду, по переменным, по методам и по классам, обрабатывает изменяемые объекты и массивы [5].

По перечисленным свойствам специализатор *CILPE* превосходит известные специализаторы для объектно-ориентированных языков [4].

Литература

1. Todd 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, pp.188–200, 2001. Springer-Verlag Berlin Heidelberg, 2001. URL: www.springerlink.com/content/p152067k1213k606/ (дата обращения: 06.04.2009).

2. Jones N.D., Gomard C.K., Sestoft P. Partial Evaluation and Automatic Compiler Generation // C.A.R. Hoare Series, Prentice-Hall, 1993.

3. Климов Ю.А. Особенности применения метода частичных вычислений к специализации программ на объектно-ориентированных языках // Препр. ИПМ им. М.В. Келдыша. 2008. № 12. 27 с. URL: http://library.keldysh.ru/preprint.asp?id=2008-12 (дата обращения: 06.04.2009).

4. Partial Evaluation for Common Intermediate Language / A.M. Chepovsky [et al.] // M. Broy and A.V. Zamulin (Eds.): PSI 2003, LNCS 2890, pp. 171–177, 2003. Springer-Verlag Berlin Heidelberg, 2003. URL: www.springerlink.com/content/r4rr30ajjn449q1a/ (дата обращения: 06.04.2009).

5. Климов Ю.А. Возможности специализатора CILPE и примеры его применения к программам на объектно-ориентированных языках // Препр. ИПМ им. М.В. Келдыша. 2008. № 30. 28 с. URL: http://library.keldysh.ru/preprint.asp?id=2008-30 (дата обращения: 06.04.2009).

SPSC: СУПЕРКОМПИЛЯТОР НА ЯЗЫКЕ SCALA

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

И.Г. Ключников; С.А. Романенко, к.ф.-м.н.

(ИПМ им. М.В. Келдыша РАН, г. Москва, ilya.klyuchnikov@gmail.com)

Суперкомпиляция является мощным и довольно сложным методом анализа и оптимизации программ. В данной статье представлен *полный* текст простого, но реально функционирующего суперкомпилятора. Это дает возможность программисту-практику ознакомиться с основными идеями и принципами суперкомпиляции, сформулированными на понятном ему языке, в виде конкретных работоспособных программ.

Ключевые слова: *преобразования программ, анализ и оптимизация программ, метавычисления, суперкомпиляция, суперкомпилятор, специализация программ, Scala, функциональное и объектно-ориентированное программирование.*

Суперкомпиляция [1–3] – метод преобразования программ, предложенный В.Ф. Турчиным в 70-х годах прошлого века. Несмотря на то, что суперкомпиляции посвящено немало литературы, практически везде алгоритмы суперкомпиляции описаны либо неполностью и фрагментарно, либо чрезмерно абстрактно (например, в виде каких-то спецификаций или правил вывода). Поэтому трудно проверить работоспособность предлагаемых методов и алгоритмов. Если информация неполная, читателю предлагается самому восстановить недостающие части (и нет гарантии, что он сделает это именно так, как подразумевал автор статьи). Если материал представлен слишком абстрактно, превращение его в работающие программы – сложный и неоднозначный процесс.

Цель данной статьи – представить *полный* текст простого, но работоспособного суперкомпилятора. Это даст возможность программисту-практику ознакомиться с суперкомпиляцией, представленной на понятном ему языке – в виде конкретных программ. Причем при желании можно проверить представленные алгоритмы на деле, взяв текст суперкомпилятора прямо из статьи.

В качестве языка реализации суперкомпилятора используется язык *Scala* [4]. Это объясняется тем, что для описания одних аспектов суперкомпиляции хорошо подходит функциональный стиль программирования, для описания других – объектно-ориентированный, а особенностью языка *Scala* является то, что в нем удалось органично соединить возможности как функционального, так и объектно-ориентированного программирования.

При этом *Scala* – вполне практичный язык программирования: все, что может быть написано на языке *Java*, легко может быть записано и на языке *Scala* (причем программа при этом становится короче). В настоящее время имеется реализация языка *Scala*, основанная на компиляции в виртуальный код *JVM* (*Java Virtual Machine*) и обеспечивающая возможность создавать программы, различные части которых написаны на *Scala* и на *Java*.

В статье рассматривается простейший суперкомпилятор *SPSC* для *ленивого* функционального языка первого порядка [5].

Рассмотрим программу на простом функциональном языке, конкатенирующую списки:

gApp(Nil(), vs)=vs;
gApp(Cons(u, us), vs)=Cons(u, gApp(us, vs));

Допустим, требуется получить результат конкатенации трех списков. Это можно сделать, вычислив выражение **gApp(gApp(xs, ys), zs)**. Однако при этом элементы списка **xs** будут анализиро-