О р д е н а   Л е н и н а
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Р о с с и й с к о й   а к а д е м и и   н а у к

Ю.А. Климов

Возможности специализатора CILPE и примеры его применения к программам на объектно-ориентированных языках

Москва
2008

Ю.А. Климов

Возможности специализатора CILPE и примеры его применения к программам на объектно-ориентированных языках

Аннотация

В работе рассматриваются возможности специализатора CILPE для объектно-ориентированного языка CIL платформы Microsoft .NET. Приводятся примеры специализации программ на объектно-ориентированном языке C#. Также рассматриваются примеры специализации результата компиляции с функциональных языков SML и Рефал в объектно-ориентированный язык CIL.

Работа поддержана проектами РФФИ № 06-01-00574-а и № 08-07-00280-а и проектом Роснауки № 2007-4-1.4-18-02-64.

Yu.A. Klimov

Specializer CILPE: examples of object-oriented program specialization

Abstract

The features of CILPE, a program specializer for Common Intermediate Language (CIL) of the Microsoft .NET platform based on Partial Evaluation (PE). Examples of object-oriented specialization, as well as specialization of the result of compilation from functional languages SML and Refal to CIL, are discussed.

Содержание

1. Введение 4
2. Специализация программ, работающих с примитивными данными 6
   2.1. Вычисление степени 6
   2.2. Специализация вычисления степени 7
   2.3. Поиск подстроки в строке 8
   2.4. Специализация функции Аккермана 10
3. Специализация программ, работающих с объектами 11
   3.1. Вычисление объектов 11
   3.2. Специализация массива 13
   3.3. Удаление избыточных объектов 15
   3.4. Виртуальные методы и арифметические выражения 17
   3.5. Специализация шаблона программирования «посетитель» 18
   3.6. Специализация нераскрываемых методов 21
4. Специализация программ на функциональных языках 24
   4.1. Специализация функции map в SML.NET 24
   4.2. Специализация Рефал-программы 25
5. Заключение 26
6. Благодарности 26
7. Литература 26

1. Введение

Оптимизация программ на основе использования априорной информации о значении части переменных называется *специализацией*. Одним из широко используемых методов специализации является метод ч*астичных вычислений* (Partial Evaluation, PE) [Jone93].

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 языках программирования, например, Ява и C#.

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.

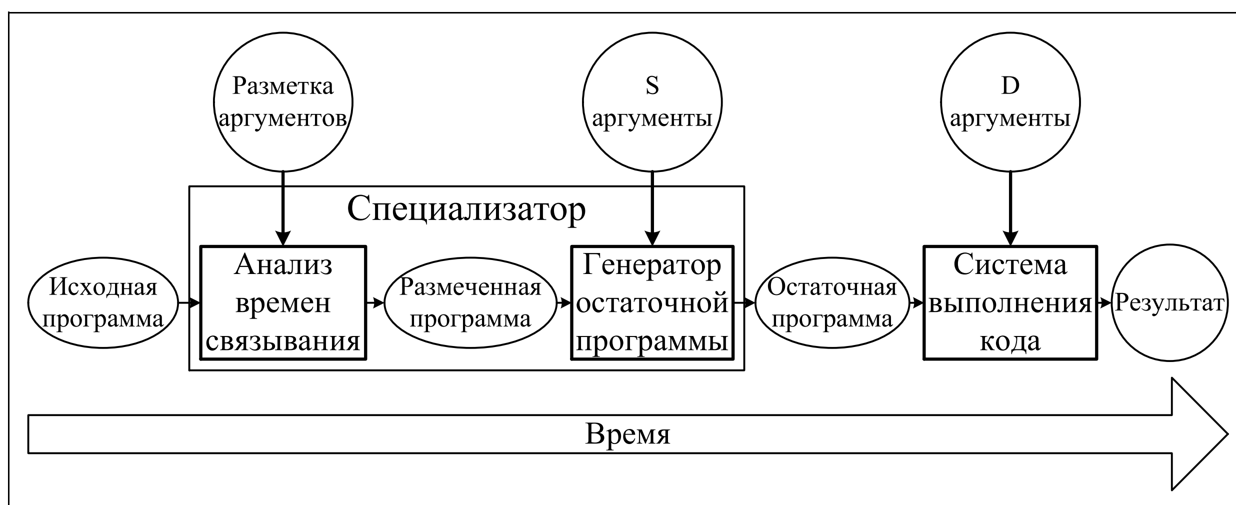

Рис. 1. Общепринятая структура специализатора, основанного на методе частичных вычислений.

Часть метода специализации, отвечающая за разделение операций и данных, называется *анализом времен связывания* (Binding Time Analysis, BTA, BT-анализ) (рис. 1). Вторая часть метода специализации, отвечающая за вычисление статической части программы и выделение динамической части в отдельную программу, называется *генератором остаточной программы* (Re-

sidual Program Generator, RPG). В этой части, собственно, и происходят «частичные вычисления».

На основе метода частичных вычислений создан специализатор CILPE для платформы Microsoft .NET [MS.NET], удовлетворяющий требованиям, изложенным в [Klim08]. В этой работе были сформулированы требования, которыми должен обладать специализатор программ на объектно-ориентированных языках.

В данной работе на примерах описаны возможности специализатора CILPE. В каждом примере продемонстрирована ситуация использования некоторой техники программирования и показаны возможности специализатора в каждом случае.

Специализатор CILPE работает с языком Common Intermediate Language (CIL) платформы Microsoft .NET [CLI]. Основным высокоуровневым языком этой платформы является объектно-ориентированный язык C#. Также на этой платформе реализовано множество других языков: SML, J# и другие.

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

Ниже приведены примеры специализации на объектно-ориентированном языке C# платформы Microsoft .NET [C#]. Эти примеры компилировались в CIL и пропускались через специализатор CILPE. Для удобства анализа результата специализации и сравнения с исходной программой в данной работе результат специализации приводится на языке C#.

Задание на специализацию формулируется с помощью атрибутов методов, которые вставляются в программу.

Атрибут *[Specialize]* указывает специализируемый метод, причем считается, что все аргументы данного метода динамические, то есть неизвестны. Специализатор строит новую версию заданного метода и заменяет исходный метод на специализированный. Также специализатор может добавить новые

методы, которые используются в специализированной версии метода.

Атрибут *[Inline]* указывает специализатору, что при специализации вызовы данного метода могут быть заменены текстами этого метода.

Все методы, не помеченные атрибутом *[Specialize],* переходят в остаточную программу без изменения. Для уменьшения размера примеров не будем приводить такие методы остаточной программы.

2. Специализация программ, работающих с примитивными данными

2.1. Вычисление степени

Рассмотрим классический пример А.П. Ершова [Ersh77] — программу возведения в степень (рис. 2). Метод *ToPower* возводит *x* в степень *n* с помощью цикла. Метод *Sample* вычисляет 2.0 в степени 5.

```
class TestClass {
  [Inline] public static double ToPower (double x, int n) {
    double r = 1.0;
    while (n != 0)
      if (n%2 == 0) {
        x = x*x; n = n/2;
      } else {
        r = r*x; n = n-1;
      }
    return r;
  }
  [Specialize] public static double Sample () { return ToPower(2.0, 5); }
}
```

Рис. 2. Исходная программа.

```
class TestClass {
  public static double Sample () { return 32.0; }
}
```

Рис. 3. Результат специализации.

В программе проставлены следующие атрибуты: атрибут *[Specialize]* у метода *Sample* указывает специализируемый метод; атрибут *[Inline]* у метода *ToPower* указывает специализатору, что вызовы данного метода нужно заме-

нить на тело метода.

В результате специализации получаем новую программу, которая отличается от старой программы заменой метода *Sample* (рис. 3). В специализированном методе *Sampl*е не осталось никаких вычислений: присутствует только оператор возврата значения.

2.2. Специализация вычисления степени

Рассмотрим немного измененный предыдущий пример (рис. 4): специализируется метод *ToPower20*, вычисляющий двадцатую степень своего аргумента.

```
class TestClass {
   [Inline] public static double ToPower (double x, int n) {
      double r = 1.0;
      while (n != 0)
         if (n%2 == 0) {
            x = x*x; n = n/2;
         } else {
            r = r*x; n = n-1;
         }
      return r;
   }
   [Specialize]
   public static double ToPower20 (double x) { return ToPower(x, 20); }
}
```

Рис. 4. Исходная программа.

```
class TestClass {
   public static double ToPower20 (double x) {
      double r;
      x = x*x; x = x*x; r = 1.0*x;
      x = x*x; x = x*x; r = r*x;
      return r;
   }
}
```

Рис. 5. Результат специализации.

В результате специализации получаем программу, состоящую из линейной последовательности операций, вычисляющих двадцатую степень ар-

гумента (рис. 5). Все проверки и операции с известной степенью *n* были выполнены специализатором. А все операции с неизвестным аргументом *x* перешли в остаточную программу.

Следует отметить, что специализатор CILPE является поливариантным по переменным и по коду: значение переменной *r* известно во время специализации до первого присваивания в эту переменную. Поэтому известное значение *1.0* не хранилось в переменной *r* остаточной программы, а непосредственно использовалось при умножении на *x*.

2.3. Поиск подстроки в строке

Рассмотрим следующий пример — поиск подстроки в строке. Ожидаем, что специализатор по известной подстроке построит эффективную программу, работающую по алгоритму Крута-Морриса-Пратта [Knut77] за линейное время от длины строки.

В «стандартной» программе сравниваются элементы строки и элементы подстроки, т.е. известные и неизвестные значения. Ожидать эффективной специализации такой программы нельзя. Изменим немного программу, чтобы в основном сравнивались элементы из известной подстроки (рис. 6).

Программа состоит из двух основных методов: *IsSubstring* и *BackTrack*. Первый метод по очереди сравнивает элементы строки и подстроки. Если на каком-то шаге элементы стали неравными, то вызывается метод *BackTrack*, который для заданной строки ищет максимальный нетривиальный префикс, который являлся бы и суффиксом этой строки. В этом методе строка задается неявно: она является началом длины *j* строки *s*. Метод *IsSubstring* возвращает длину такого префикса.

В результате специализации получаем программу (рис. 7) — конечный автомат — которая один раз проходит по строке, проверяя, содержит ли строка-аргумент заданную подстроку.

В результате получаем снижение сложности вычисления примерно в 3.5 раза для примера, приведенного выше.

```
class TestClass {
   public static int D (int x) { return x; }
   [Inline] public static int BackTrack (string s, int j) {
      int k;
      for (k = j-1; k > 0; k--) {
         int i;
         for (i = 0; i < k; i++) if (s[i] != s[j-k+i]) break;
         if (i == k && s[j] != s[k]) break;
      }
      return k;
   }
   [Inline] public static bool IsSubstring (string s, string x) {
      int j = 0;
      for (int i = D(0); i < x.Length; i++) {
         while (x[i] != s[j]) { if (j == 0) continue; j = BackTrack(s, j); }
         j++;
         if (j == s.Length) return true;
      }
      return false;
   }
   [Specialize] public static bool Test (string d) {
      return IsSubstring("aaaab", d);
   }
}
```

Рис. 6. Исходная программа.

```
class TestClass {
   public static int D_0 () { return 0; }
   public static bool Test (string x) {
         int i = D_0();
   L0:  if (i >= x.Length) return false;
         if (x[i++] != 'a') goto L0;
         if (i >= x.Length) return false;
         if (x[i++] != 'a') goto L0;
         if (i >= x.Length) return false;
         if (x[i++] != 'a') goto L0;
         if (i >= x.Length) return false;
   L1:  if (x[i++] != 'a') goto L0;
         if (i >= x.Length) return false;
         if (x[i] != 'b') goto L1;
         return true;
}  }
```

Рис. 7. Результат специализации.

2.4. Специализация функции Аккермана

В предыдущих примерах все функции были помечены *[Inline]*, что позволяло разворачивать функции во время специализации. Но если глубина рекурсии неизвестна во время специализации, то такое разворачивание будет продолжаться бесконечно долго.

Рассмотрим следующий пример — программу, вычисляющую функцию Аккермана (рис. 8). В специализированном методе вызовем функцию Аккермана с известным первым аргументом, равным *3*.

```
class TestClass {
   public static ulong A (ulong x, ulong y) {
      if (x == 0) return y+1;
      else if (y == 0) return A(x-1, 1);
      else return A(x-1, A(x, y-1));
   }
   [Specialize] public static ulong Test (ulong y) { return A(3, y); }
}
```

Рис. 8. Исходная программа.

```
class TestClass {
   public static ulong A_3 (ulong y) { // x = 3
      return y == 0 ? A_2_1() : A_2(A_3(y-1));
   }
   public static ulong A_2 (ulong y) { // x = 2
      return y == 0 ? A_1_1() : A_1(A_2(y-1));
   }
   public static ulong A_1 (ulong y) { // x = 1
      return y == 0 ? A_0_1() : A_0(A_1(y-1));
   }
   public static ulong A_0 (ulong y) { return y+1; } // x = 0
   public static ulong A_2_1 () { return A_1(A_2_0()); } // x = 2, y = 1
   public static ulong A_2_0 () { return A_1_1(); } // x = 2, y = 0
   public static ulong A_1_1 () { return A_0(A_1_0()); } // x = 1, y = 1
   public static ulong A_1_0 () { return A_0_1(); } // x = 1, y = 0
   public static ulong A_0_1 () { return 2; } // x = 0, y = 1
   public static ulong Test (ulong y) { return A_3(y); }
}
```

Рис. 9. Результат специализации.

Метод *A* не помечен *[Inline]*, поэтому специализатор не может подставить тело этого метода в точке вызова. Поэтому в результате специализации получаем (рис. 9) несколько проспециализированных версий данного метода. Раскрытие методов и отсутствие операций с первым, а иногда и со вторым аргументами позволяет вычислять функцию более быстро. Например, ускорение вычисления функции Аккермана *A(3,10)* составило около 5 раз.

Отметим, что проспециализированные версии отличаются не только значением известного аргумента (для которого они были сгенерированы), но и набором известных аргументов. Методы *A_1*, *A_2*, *A_3* были получены для известного значения первого аргумента *x* и неизвестного значения второго аргумента *y*. А методы *A_0_1*, *A_1_0*, *A_1_1*, *A_2_0*, *A_2_1* — для известных значений и первого и второго аргументов.

Этот пример показывает, что CILPE — поливариантный по методам специализатор: в зависимости от использования, от известности или неизвестности аргументов, получаются специализированные версии методов с разным набором параметров.

3. Специализация программ, работающих с объектами

3.1. Вычисление объектов

В предыдущих примерах присутствовали только операции над примитивными данными и строками. Этот пример показывает работу специализатора с полностью известными данными, в том числе и объектами, операции над которыми полностью выполняются во время специализации. В частности, если у метода с атрибутом *[Specialize]* нет аргументов и побочных эффектов, а также возвращаемое значение имеет примитивный тип (*int*, *double*, …) или тип строки (*string*), то этот метод будет заменен методом, просто возвращающий значение.

Часть библиотечных методов обладают побочным эффектом: скажем, печать на экран или, наоборот, чтение данных из файла. Мы ожидаем, что эти операции не будут выполнены во время специализации и перейдут в остаточ-

ную программу. Другие же методы, наоборот, не обладают побочным эффектом и могут быть выполнены во время специализации программы.

```
public class Container {
   public Container prev;
   public Container next;
   public int val;
   [Inline] public Container () { }
}
public class FibTest {
   [Inline] static public Container Create (int n) {
      Container first = new Container();
      Container last = first;
      for (int i = 1; i < n; i++) {
         last.next = new Container();
         last.next.prev = last;
         last = last.next;
      }
      return first;
   }
   [Inline] static public void Calculate (Container con) {
      if (con == null) return;
      con.val = 0;
      con = con.next;
      if (con == null) return;
      con.val = 1;
      con = con.next;
      for ( ; con != null; con = con.next)
         con.val = con.prev.val + con.prev.prev.val;
   }
   [Specialize] static public string Test ()
      Container con = Create(11);
      Calculate(con);
      string res = null;
      for ( ; con != null; con = con.next)
         res = (res == null ? "" : res + " ") + con.val;
      return res;
   }
}
```

Рис. 10. Исходная программа.

Если специализируемый метод и программа в целом удовлетворяют следующим условиям, то специализатор полностью его вычислит, оставив в

остаточной программе только оператор возврата значения:

1. Метод не имеет аргументов, возвращаемое значение имеет примитивный тип (*int*, *double*, …) или тип строки (*string*).
2. Про все используемые в программе библиотечные методы указано, что они не имеют побочных эффектов.
3. Про все методы указано, что специализатор имеет право их развернуть (указан атрибут *[inline]*).

В качестве примера рассмотрим программу, вычисляющую последовательность чисел Фибоначчи до десятого элемента (рис. 10). Вначале создается двусвязный список из объектов класса Container, потом в эти объекты записываются числа Фибоначчи.

Вычисление чисел Фибоначчи запрограммировано таким неэффективным образом, чтобы показать, что специализатор вычислит все объекты, и в результате получается метод, возвращающий строку из чисел Фибоначчи (рис.11).

```
public class FibTest {
   static public string Test () { return "0 1 1 2 3 5 8 13 21 34 55"; }
}
```

Рис. 11. Результат специализации.

3.2. Специализация массива

Массивы широко используются в объектно-ориентированных языках, поэтому должны эффективно поддерживаться специализатором.

Рассмотрим пример вычисления значения многочлена по схеме Горнера (рис. 12). Класс *Polynomial* представляет собой многочлен. Коэффициенты многочлена хранятся в массиве *coef*. Метод *Calc(double x)* вычисляет значение многочлена в точке x.

Атрибутом *[Specialize]* помечены два метода: *TestS* и *TestD*. В методе *TestS* коэффициенты массива заданы явно, а переменная x — параметр метода. В методе *TestD* коэффициенты массива, как и переменная x, передаются через параметры.

```
class Polynomial {
   double[] coef;
   [Inline] public Polynomial (double[] coef) { this.coef = coef; }
   [Inline] public double Calc (double x) {
      double res = 0;
      for (int i = coef.Length-1; i >= 0; i--) res = res*x+coef[i];
      return res;
   }
}
class TestClass {
   [Specialize] public static double TestS (double x) {
      double[] coef = {1, 2, 3}; // p(x) = 1+2x+3x^2
      Polynomial p = new Polynomial(coef);
      return p.Calc(x);
   }
   [Specialize]
   public static double TestD (double a0, double a1, double a2, double x) {
      double[] coef = {a0, a1, a2}; // p(x) = a0+a1*x+a2*x^2
      Polynomial p = new Polynomial(coef);
      return p.Calc(x);
   }
}
```

Рис. 12. Исходная программа.

```
class TestClass {
   public static double TestS (double x) {
      return ((0*x+3)*x+2)*x+1;
   }
   public static double TestD (double a0, double a1, double a2, double x) {
      return ((0*x+a2)*x+a1)*x+a0;
   }
}
```

Рис. 13. Результат специализации.

В результате специализации обоих методов получаем арифметические выражения, вычисляющие значение многочлена (рис. 13). В методе *TestS* остается выражение только от аргумента *x*, а в примере *TestD* — выражение, вычисляющее значения многочлена второй степени через коэффициенты *a0*, *a1* и *a2* и переменную *x*.

Объект класса *Polynomial* и массив коэффициентов *coef* известны во

время специализации программы и полностью вычисляются специализатором, Так как длина массива в обоих случаях известна цикл в методе *Calc* полностью разворачивается, и операции над переменной *i* выполняются.

В остаточную программу переходят только операции над параметром *x* в первом случае и операции над параметром x и значениями коэффициентов массива *a0*, *a1* и *a2* во втором случае.

3.3. Удаление избыточных объектов

В некоторых случаях объекты создаются для временного хранения или передачи данных. В предыдущих двух примерах создавался один такой «временный» объект.

В следующем примере (рис. 14) в специализируемом методе *Test* в цикле создаются несколько (в данном случае 3) временных объектов, которые используются ниже в теле этого метода.

```
class Container {
   int n;
   double x, y;
   [Inline] public Container (int n, double x) {
      this.n = n; this.x = x; this.y = this.n*this.x;
   }
   [Inline] public double GetX () { return x; }
   [Inline] public double GetY () { return y; }
}
class TestClass {
   [Specialize] public static double Test (double x) {
      Container[] cs = new Container[3];
      for (int i = 0; i < cs.Length; i++) cs[i] = new Container(i+1, x);
      double sx = 0.0;
      for (int i = 0; i < cs.Length; i++) sx += cs[i].GetX();
      double sy = 0.0;
      for (int i = 0; i < cs.Length; i++) sy += cs[i].GetY();
      return sx/sy;
   }
}
```

Рис. 14. Исходная программа.

В результате специализации получаем программу, не содержащую ин-

струкций создания объектов класса *Container* (рис. 15).

```
class TestClass {
   public static double Test (double x) {
      container1_x = x; // cs[0] = new Container(1, x);
      container1_y = 1*x;
      container2_x = x; // cs[1] = new Container(2, x);
      container2_y = 2*x;
      container3_x = x; // cs[2] = new Container(3, x);
      container3_y = 3*x;
      double sx = 0.0+container1_x+container2_x+container3_x;
      double sy = 0.0+container1_y+container2_y+container3_y;
      return sx/sy;
   }
}
```

Рис. 15. Результат специализации.

Каждый объект класса *Container* обладает тремя полями — *n*, *x* и *y*. Значение поля n будет известно во время специализации, а значения полей *x* и *y* — нет.

Поэтому в остаточной программе вместо заведения объектов класса *Container* заведены локальные переменные для каждого поля для каждого создания объекта: переменные *container1_x* и *container1_y* заведены для объекта, который создается при первом проходе цикла, переменные *container2_x* и *container2_y* — при втором, *container3_x* и *container3_y* — при третьем проходе цикла. А для поля *n* переменные не заводятся, т.к. значения этого поля известны во время специализации и подставлены непосредственно при вычислении переменных container1_y, container2_y и container3_y.

Отметим, что переменные заводятся для каждого *создания* объекта, а не для каждого оператора *new* в исходной программе.

Таким образом, все неизвестные поля известных объектов переходят в локальные переменные. А известные поля обрабатываются аналогично известным локальным переменным — их значения вычисляются и подставляются при вычислении значений неизвестных переменных.

Этот и предыдущие примеры показывают, что когда создание и ис-

пользование объекта или массива полностью прослеживается, специализатор CILPE может создать и использовать объект или массив во время специализации. В остаточную программу не переходят операции создания и использования объекта. Если поля объекта или элементы массива неизвестны, то они заменяются локальными переменными. А если известны — то аналогично самому объекту вычисляются во время специализации.

Тем самым, специализатор удаляет избыточные объекты и промежуточные данные, что близко к идеям и результатам deforestation [Wadl88].

3.4. Виртуальные методы и арифметические выражения

В рассмотренных выше случаях методы были невиртуальными или статическими: всегда известно тело вызываемого метода. Специализатор CILPE объектно-ориентированного языка поддерживает виртуальные методы, тела которых определяются по точному типу объекта.

Отметим, что на втором этапе специализации, на этапе генерации остаточной программы, известны типы всех известных объектов, поэтому всегда можно точно определить вызываемый метод. Но во время анализа времен связывания, точный тип неизвестен, а, следовательно, неизвестно и тело метода при виртуальном вызове. В этом случае анализ времен связывания должен разобрать все возможные типы значений переменных.

Рассмотрим программу, в которой из объектов конструируется выражение, которое затем вычисляется (рис. 16).

В программе описан абстрактный класс *Expr*, у которого определен виртуальный метод *GetValue* для вычисления значения выражения. Также определены два подкласса класса *Expr*: для операции сложения и для константы и переменной. Можно определить подклассы и для других арифметических операций, но для примера достаточно этих двух подклассов.

В методе *Test* в цикле создается конфигурация из объектов, соответствующая выражению $(x+1)+2$, потом в объект-переменную x записывается значение переменной и выражение вычисляется.

```
abstract class Expr {
   [Inline] public Expr () { }
   [Inline] public abstract double GetValue ();
}
class Plus : Expr {
   Expr x, y;
   [Inline] public Plus (Expr x, Expr y) { this.x = x; this.y = y; }
   [Inline] public override double GetValue () {
      return x.GetValue()+y.GetValue();
   }
}
class Value : Expr {
   double x;
   [Inline] public Value () { this.x = 0; }
   [Inline] public Value (double x) { this.x = x; }
   [Inline] public void SetValue (double x) { this.x = x; }
   [Inline] public override double GetValue () { return x; }
}
class TestClass {
   [Specialize] public static double Test (double x) {
      Value var = new Value();
      Expr expr = var;
      for (int i = 1; i < 3; i++) expr = new Plus(expr, new Value(i));
      var.SetValue(x);
      return expr.GetValue();
} }
```

Рис. 16. Исходная программа.

```
class TestClass {
   public static double Test (double x) { return (x+1)+2; }
}
```

Рис. 17. Результат специализации.

В результате специализации получаем остаточную программу, в которой отсутствуют операции создания объектов, а присутствует формула для вычисления выражения $(x+1)+2$ (рис. 17).

3.5. Специализация шаблона программирования «посетитель»

Одним из часто используемых подходов к программированию является шаблон «посетитель» (visitor pattern). «Посетители» используются, чтобы избежать разбора случаев в зависимости от типа аргумента.

```
abstract class Visitor {
   [Inline] public Visitor () {}
   [Inline] public abstract void visit (Car car);
   [Inline] public abstract void visit (Wheel wheel); }
abstract class Visitable {
   [Inline] public Visitable () {}
   [Inline] public abstract void accept (Visitor visitor); }
class Car : Visitable {
   Visitable[] parts;
   [Inline] public Car () {
      parts = new Visitable[] { new Wheel("front left"), new Wheel("front right"),
                                new Wheel("back left"), new Wheel("back right") }; }
   [Inline] public override void accept (Visitor visitor) {
      visitor.visit(this);
      for (int i = 0; i <parts.Length; i++) parts[i].accept(visitor); } }
class Wheel : Visitable {
   string name;
   [Inline] public Wheel (string name) { this.name = name; }
   [Inline] public string getName () { return.name; }
   [Inline] public override void accept (Visitor visitor) { visitor.visit(this); } }
class PrintVisitor : Visitor {
   [Inline] public PrintVisitor () {}
   [Inline] public override void visit (Car car) {
      Console.WriteLine("Visiting car"); }
   [Inline] public override void visit (Wheel wheel) {
      Console.WriteLine("Visiting {0} wheel", wheel.getName()); } }
class CountVisitor : Visitor {
   int carCount, wheelCount;
   [Inline] public CountVisitor () { carCount = 0; wheelCount = 0; }
   [Inline] public override void visit (Car car) {
      Console.WriteLine("Visiting {0} car", carCount++); }
   [Inline] public override void visit (Wheel wheel) {
      Console.WriteLine("Visiting {0} wheel", wheelCount++); } }
public class VisitorPatternTest {
   [Inline] static Visitor CreateVisitor (bool p) {
      return p ? (Visitor) new CountVisitor() : new PrintVisitor(); }
   [Inline] static Visitable CreateObject (bool q) {
      return q ? (Visitable) new Car() : new Wheel("one"); }
   [Specialize] static public void Test (bool p, bool q) {
      Visitor vis = CreateVisitor(p);
      Visitable obj = CreateObject(q);
      obj.accept(vis); } }
```

Рис. 18. Исходная программа.

```
class VisitorPatternTest {
   static public void Test (bool p, bool q) {
      if (p) {
         if (q) {
            Console.WriteLine("Visiting {0} car", 0);
            Console.WriteLine("Visiting {0} wheel", 0);
            Console.WriteLine("Visiting {0} wheel", 1);
            Console.WriteLine("Visiting {0} wheel", 2);
            Console.WriteLine("Visiting {0} wheel", 3);
         } else
            Console.WriteLine("Visiting {0} wheel", 0);
      } else {
         if (q) {
            Console.WriteLine("Visiting car");
            Console.WriteLine("Visiting {0} wheel", "front left");
            Console.WriteLine("Visiting {0} wheel", "front right");
            Console.WriteLine("Visiting {0} wheel", "back left");
            Console.WriteLine("Visiting {0} wheel", "back right");
         } else
            Console.WriteLine("Visiting {0} wheel", "one");
      }
   }
}
```

Рис. 19. Результат специализации.

Пример взят со страницы http://en.wikipedia.org/wiki/Visitors_pattern (рис. 18). В этом примере описывается абстрактный класс *Visitor* — класс, описывающий методы для каждого потомка класса *Visitable*, набор наследников класса *Visitable*, представляющих собой машину, двигатель, кузов и колеса. И два наследника класса *Visitor* — один для печати списка деталей, другой для их подсчета. Тест состоит в вызове для созданного объекта одного из визитеров.

В этом примере специализируемый метод ничего не возвращает, но вызывает функцию с побочным эффектом Console.WriteLine, которая выдает строку на экран. Специализатор CILPE все функции с побочными эффектами переносит в остаточную программу. Иначе, в данном примере печать происходила бы не во время исполнения остаточной программы, как это ожидается,

а во время специализации.

В результате специализации получаем программу, проводящую проверки аргументов и выводящую результат на печать (рис. 19). В остаточной программе не содержатся создания объектов и вызовы методов.

3.6. Специализация нераскрываемых методов

В предыдущих примерах методы раскрывались (атрибут *[Inline]*): их тела подставлялись в вызывающий метод. Но во многих случаях это нецелесообразно или невозможно: например, в случае рекурсии такое раскрытие может продолжаться бесконечно долго.

Раскрывать или не раскрывать метод — это алгоритмически неразрешимая задача: невозможно понять, будет ли раскрытие метода продолжаться бесконечно долго. Поэтому программист должен указать те методы, которые могут быть раскрыты атрибутом *[Inline]*, а остальные методы считаются нераскрываемыми.

Если не указан атрибут *[Inline]*, то по исходному методу специализатор сгенерирует один или несколько методов:

1. Если в нераскрываемый метод в качестве аргумента передается известный объект, то вместо передачи объекта в метод в остаточной программе должны передаваться *по ссылке* локальные переменные, созданные для полей этого объекта.
2. Специализатор по значениям известных аргументов метода должен сгенерировать специализированную версию исходного метода. Поэтому в остаточной программе может получиться несколько методов из одного.

В следующем примере (рис. 20) описан класс *Container* с полем *n*. Статический метод *Increase* увеличивает поле *n* первого аргумента *c* на второй аргумент *i* и печатает новое значение этого поля.

В специализируемом методе *Test* создается два объекта класса *Container*, для них вызываются метод *Container* и второй объект возвращается.

```
class Container {
   public int n;
   [Inline] public Container (int n) { this.n = n; }
}
class TestClass {
   public static void Increase (Container c, int i) {
      c.n += i; Console.WriteLine("{0}", c.n);
   }
   [Specialize] public static Container Test (int n) {
      Container cs0 = new Container(n);
      Container cs1 = new Container(0);
      Increase(cs0, 5);
      Increase(cs0, n);
      Increase(cs1, 5);
      return cs1;
   }
}
```

Рис. 20. Исходная программа.

```
class Container {
   public int n;
   public Container () { this.n = 0; } // n = 0
}
class TestClass {
   public static void Increase_n_5 (ref int c_n) { // i = 5
      c_n += 5; Console.WriteLine("{0}", c_n);
   }
   public static void Increase_n (ref int c_n, int i) {
      c_n += i; Console.WriteLine("{0}", c_n);
   }
   public static void Increase_5 (Container c) { // i = 5
      c.n += 5; Console.WriteLine("{0}", c.n);
   }
   public static Container Test (int n) {
      int cs0_n = n;
      Container cs1 = new Container();
      Increase_n_5(ref cs0_n);
      Increase_n(ref cs0_n, n);
      Increase_5(cs1);
      return cs1;
   }
}
```

Рис. 21. Результат специализации.

В результате специализации (рис. 21) получаем программу, содержащую создание только одного объекта класса *Container*. И вместо универсального метода *Inscrease* вызываются его специализированные версии.

Объект, записанный в переменную *cs0*, известен во время специализации, поэтому он полностью вычисляется и не переходит в остаточную программу. Но поле *n* этого объекта неизвестно, поэтому вместо поля для объекта заводится локальная переменная *cs0_n*, которая передается по ссылке в методы *Increase*.

Отметим, что объекты класса *Container cs0* и *cs1* в исходной программе хоть и описаны одинаково, но используются по разному: первый известен и вычисляется специализатором, второй переходит в остаточную программу. Это показывает, что специализатор CILPE обладает поливариантностью по классам.

В исходной программе одна и та же часть кода — метод Increase — используется в различных ситуациях: в одном случае объект некоторого класса используется для временного хранения данных, в другом — с помощью него данные возвращаются из специализированного метода.

В таких ситуациях специализатор генерирует остаточные методы для каждого случая. Это означает, что он обладает поливариантностью по методам.

В приведенном примере создается три специализируемых версии метода *Increase*:

1. Метод *Increase_n_5* — специализированная версия для известного объекта и известного второго аргумента, равного *5*.
2. Метод *Increase_n* — специализированная версия для известного объекта и неизвестного второго аргумента.
3. Метод *Increase_5* — специализированная версия для неизвестного объекта и известного второго аргумента, равного *5*.

4. Специализация программ на функциональных языках

Объектно-ориентированные языки являются основой современных многоязычных платформ MS.NET и Java. Для этих платформ существуют компиляторы различных языков программирования. Поэтому специализатор CILPE может быть применен к программам на языках программирования, реализованных для MS.NET. Примером таких языков являются функциональные языки SML и Refal и объектно-ориентированный язык J#.

4.1. Специализация функции map в SML.NET

Рассмотрим пример на языке SML, на котором описана функция высшего порядка map — применение функции (первый аргумент) к каждому элементу списка (второй аргумент) (рис. 22). Специализируемая функция *test* вызывает функцию map с известным первым аргументом — функцией увеличения аргумента на *1*.

Перед специализацией исходная программа на SML компилируется в объектно-ориентированный язык CIL — внутренний язык платформы MS.NET. Специализатор обрабатывает этот язык и выдает результат на том же языке CIL. Однако, для сопоставления исходной программы и результата специализации результат специализации представлен на языке SML.

В результате специализации получаем специализированную версию функции *map* — функцию *map_1*, которая увеличивает на 1 все элементы списка. Ее эквивалентное представление на SML показано на рис. 23.

```
fun map f []        = []
|   map f (x::xs) = (f x)::(map f xs)

fun test xs = map (fn x => x+1) xs
```

Рис. 22. Исходная программа на SML.

```
fun map_1 []        = []
|   map_1 (x::xs) = (x+1)::(map_1 xs)

fun test xs = map_1 xs
```

Рис. 23. Результат специализации, представленный программой на SML.

В рассмотренном примере функция *(**fn** x => x+1)* при компиляции в CIL представляется объектом. Специализатор прослеживает использование этого объекта и полностью вычисляет его во время специализации. В остаточной программе не остается этого объекта, а прибавление единицы явно производится в теле метода *map_1*.

4.2. Специализация Рефал-программы

Рассмотрим пример программы на функциональном языке Рефал (рис. 24). В данном примере функция *ExprInt* разбирает и вычисляет арифметическое выражение. В этом выражении могут присутствовать: символ *'x'*, обозначающий переменную, числа, скобки и знаки сложения и умножения. В функции *ExprInt* разбираются все эти пять случаев, и в зависимости от ситуации выполняются необходимые действия.

```
ExprInt eExpr sX = eExpr : {
   'x' = sX;
   sN = sN;
   (eExpr1) = <ExprInt eExpr1 sX>;
   eA'+'eB = <ADD <ExprInt eA sX> <ExprInt eB sX>>;
   eA'*'eB = <MUL <ExprInt eA sX> <ExprInt eB sX>>;
};
Expr = 1 '+x*' ( 2 '+x*' ( 3 '+x*' ( 4 '+x*' ( 5 '+x' ))));
Test sX = <ExprInt <Expr> <FILTER_INT sX>>;
```

Рис. 24. Исходная программа на Рефале.

В этом примере специализируется функция *Test* с одним аргументом. Она вызывает функцию *ExprInt* с двумя аргументами: результатом работы функции *Expr* и результатом работы функции *FILTER_INT*. Функция *Expr* возвращает выражение в виде последовательности символов, чисел и правильно расставленных скобок (последовательность символов на Рефале записывается в одинарных кавычках). Функция *FILTER_INT* проверяет, что аргумент является числом.

Программа на Рефале сначала компилируется в язык Java. Для работы программы кроме результата компиляции еще требуются библиотеки, кото-

рые также реализованы на Java. В дальнейшем результат компиляции примера и библиотеки компилируются в язык CIL. С этим результатом и работает специализатор CILPE.

В результате компиляции получаем линейную программу на языке CIL, которая может быть представлена на языке C# следующим образом (рис. 25).

```
class Test {
   public static int Test (int x) {
      return 1+x*(2+x*(3+x*(4+x*(5+x))));
   }
}
```

Рис. 25. Результат специализации.

5. Заключение

В данной работе приведены примеры специализации программ на объектно-ориентированном языке C# и примеры на функциональных языках SML и Refal. Эти примеры показывают, что специализатор CILPE обладает широкими возможностями по специализации реальных программ:

1. Специализатор CILPE обладает поддержкой необходимого набора конструкций языка.
2. Специализатор CILPE является поливариантным по коду, по переменным, по методам и по классам.

6. Благодарности

Автор выражает благодарность участникам Рефал-семинара за обсуждение идей, Андрею Мищенко и Сергею Скоробогатову за помощь в создании специализатора CILPE, Microsoft Research за поддержку разработки в 2002-2003 годах.

7. Литература

[C#] C# programming language // http://msdn.microsoft.com/vcsharp/.

[Chep03] 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" // Manfred Broy and Alexandre V. Zamulin (eds.), Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI'2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003. LNCS 2890, Springer-Verlag, December 2003, pp.171-177.

[CLI] Common Language Infrastructure (CLI) // http://www.ecma-international.org/publications/standards/Ecma-335.htm, http://msdn2.microsoft.com/en-us/netframework/aa569283.aspx.

[Ersh77] А.П.Ершов "О сущности трансляции" // Программирование, 1977, №5, с.21-39.

[Jone93] N.D.Jones, C.K.Gomard, P.Sestoft "Partial Evaluation and Automatic Compiler Generation" // C.A.R. Hoare Series, Prentice-Hall, 1993.

[Klim05a] Ю.А.Климов "Поливариантный анализ времен связывания в специализаторе CILPE для Common Intermediate Language платформы Microsoft.NET" // Труды Всероссийской конференции "Технологии Microsoft в теории и практике программирования", Москва, 17-18 февраля 2005 г., Изд-во МГТУ им. Н.Э. Баумана, 2005, с.128.

[Klim05b] Ю.А.Климов "О поливариантном анализе времен связывания в специализаторе объектно-ориентированного языка" // Труды Всероссийской научной конференции "Научный сервис в сети Интернет: технологии распределенных вычислений", Новороссийск, 19-24 сентября 2005 г., Изд-во МГУ, с.89-91.

[Klim06] Ю.А.Климов "Генератор остаточной программы и корректность специализатора объектно-ориентированного языка" // Труды Всероссийской научной конференции "Научный сервис в сети Интернет: технологии параллельного программирования", Новороссийск, 18-23 сентября 2006 г., Изд-во МГУ, с.137-140.

[Klim08] Ю.А.Климов "Особенности применения метода частичных вычислений к специализации программ на объектно-ориентированных" // Препринт Института прикладной математики им. М.В. Келдыша РАН, 2008 г., №12.

[Knut77] D.E.Knuth, J.H.Morris, V.R.Pratt "Fast Pattern Matching in Strings" // In SIAM Journal on Computing, 6(2), 1977, pp. 323-350.

[MS.NET] Microsoft .NET Framework // http://www.microsoft.com/net/.

[Refal] Refal programming language // http://refal.ru/.

[SML] Standard ML programming language // http://www.itu.dk/~sestoft/mosml.html, http://www.smlnj.org/.

[Wadl88] P.Wadler "Deforestation: Transforming Programs to Eliminate Trees" // In European Symposium on Programming, Lecture Notes in Computer Science, Springer-Verlag, 1988.