E10-2004-135

Cs. Török*, M. Révayová**, A. Matejciková***

GENERICS-BASED VECTORIZATION IN MS.NET ROTOR

*E-mail: torokcs@tuke.sk
**E-mail: martina.revayova@tuke.sk
***E-mail: andrea.matejcikova@tuke.sk

Торок Ч., Реваева М., Матейчикова А. E10-2004-135
Векторизация на основе Generics в MS.NET Rotor

В настоящее время компания «Microsoft» разрабатывает новые программные технологии в рамках .NET. В основе .NET находятся система Common Language Infrastructure и новый язык программирования MS Visual C#. Работа посвящена разработке компонентов для векторного программирования на основе параметрического полиморфизма. Она описывает полиморфический vector-класс, созданный в MS SSCLI на базе программных модулей Gyro в MS Visual C#. В этот generic-класс входят перегрузка операций, indexer, изменение типов и некоторые другие основные функции. Вычислитель типа реализуется через интерфейс. Приводится также обсуждение его эффективности.

Работа выполнена в Лаборатории информационных технологий ОИЯИ.

Сообщение Объединенного института ядерных исследований. Дубна, 2004

Török Cs., Révayová M., Matejciková A. E10-2004-135
Generics-based Vectorization in MS.NET Rotor

Recently, Microsoft has been developing new programming technologies in the framework of .NET. The Common Language Infrastructure and the new programming language MS Visual C# lie at the center of .NET. The paper is devoted to design and implementation for support of vectorial programming based on parametric polymorphism. It describes a polymorphic vector class, created in MS SSCLI based on Gyro with MS Visual C#. The implementation of generic class contains operator overloading, indexer, casting and some other basic functionalities. The type evaluator is realized through an interface evaluator. Discussion on its performance is given too.

The investigation has been performed at the Laboratory of Information Technologies, JINR.

Communication of the Joint Institute for Nuclear Research. Dubna, 2004

INTRODUCTION

Vectorization enables one to create a program code clearer than coding based only on loops. In the 1990s many researchers that needed computations in their work turned from Fortran or Pascal to such systems as SPlus, SAS and Matlab. The main benefit of these and similar systems is the support of vectorial/matrix programming (with a rich set of functions and toolboxes). However, the drawback of coding in these systems is connected with code reuse. Therefore, many of them turned to Visual Basic or C++.

One of the reasons why we dropped Visual Basic and turned to C++ and afterward to C# is the fact that VB does not support operator overloading, a language feature, that enables one to write vector/matrix expressions (vectorization, vectorial programming) like

$$C = 2^* A^* x - 3^* \sin(B^* y),$$

where the variables represent vectors and matrices. The main object-oriented features, inheritance and polymorphism, are implemented to both C# and the new .NET version 7 of VB. However, the latter, unlike C# and VC++, miss operator overloading. So far C# 1 lacks some important features too. To avoid code duplicates in creating similar types (for example float, double or string vectors) you can not leverage the concept of polymorphic programming (also known as generics, templates). It is not part of C# by now; however, the main architect of C# A. Hejlsberg promised in 2000 that it will be included into the higher versions. Gyro [1] as part of Rotor [2] supports generics.

Rotor

The MS Common Language Runtime (CLR) is Microsoft's commercial implementation of the MS Common Language Infrastructure (CLI) specification. MS CLI provides a set of specifications for *executable code* and the *execution environment* in which it runs. The MS Shared Source Common Language Infrastructure (SS CLI) Implementation, known as Rotor, is a complete implementation of the ECMA-334 (C#) and ECMA-335 (CLI) standards in million source code lines form (compilers, memory management, JIT code generators, component infrastructure, etc.). MS SSCLI is derived from the source code of the MS CLR (CLR is the core runtime engine in the Microsoft .NET Framework for executing applications) and it is free [2].

Generics

Generics refer to classes and methods that work uniformly on values of different types. It is an extension to the CLR's type system that allows developers to define types for which certain details are left unspecified. These details are specified when the code is referenced by consumer code. Gyro is a set of files to support generic type definitions and generic methods through modifying Microsoft Shared Source CLI 1.0.

The paper [3] showed the basic techniques of the process of vectorization in C#. In [4] the LinAlg component library is presented that has been developed at the authors' department. LinAlg enables vectorial programming and incorporates numerical, statistical, graphical, and database methods. Our aim is to redesign and reimplement LinAlg based on parametric polymorphism. This paper presents the first steps sto achieving this goal.

Section 1 shows the basic work with one-dimensional arrays, generics, and the design of the base generic vector class. Section 2 is devoted to the construction of the parametric class `Vector<T>` and introduces several constructors for instantiating objects of the new class. The following section implements some useful functions. Section 4 shows how to print the whole vector based on overriding of `ToString()` method and how to provide access to the vector elements via indexer. Then we show shortly how the operator overloading works. Creating user-defined operators via operator overloading enables one to write vector arithmetic expressions. Sections 6 and 7 deal with casting between vectors and arrays, and the implementation of the vector counterparts of mathematical functions, respectively. The results of differences in performance of generic and non-generic vector classes will be presented in the last but one section.

Requirements: to run the code you need to install Perl [5], the Shared Source Common Language Infrastructure (SSCLI) [2] and Gyro [3]. You can download the code discussed in the paper from http://svfweb.tuke.sk/pracoviska/km/torok/04Publications.html#DOTNET

1. ARRAYS AND GENERIC VECTORS

In this section, we show briefly how to declare an array of float type and how to instantiate arrays as .NET Framework `System.Array` types. Then we present the main design and implementation steps in creating generic vectors.

Arrays

The declaration and instantiation of one-dimensional array can be done in either one or two steps. To create a one-dimensional array `x` of double values declare it

```
double[] x;
```

and initialize its three elements (the elements are set by default to zero):

```
   x = new double[3];
```

The same can be written with one code line:

```
   double[] x = new double[3];
```

You can use indices to work with the array elements and change their values:

```
   x[0] = 1.2;
```

Curly brackets enable a very useful way of array declaration and initialization with concrete values:

```
   float[] y = {1f, 2.1F, (float)3};
```

Class `Array` serves as the base class for all arrays in the common language runtime and provides methods for creating, manipulating, searching and sorting arrays. The main object and static methods for arrays are:

```
Length, GetLength(), Rank(), Sort(), Reverse(), BinarySearch().
```

The number of methods is not so much and what is more important, they do not support vectorial computation. However, the .NET Framework has the tools to design vector and matrix data structures that are equipped with a wide range of functions for handling various numerical, statistical, and graphical tasks.

Generics and Vectors

Managing of almost identical classes (such as `VectorFloat` or `VectorDouble` classes) is tedious, and to make the most of code reuse, it is desirable to build these data components based not only on inheritance but also on generics.

Parametric polymorphism allows parametric or generic types (classes, interfaces, and structs) and method definitions. To understand the fundamentals of generics, you can consult the papers [6, 7]. We only give here for illustration one code line of generic defining code and two code lines of generic referencing code for a parameterized class `ClassName`, and its `int` and `long` objects, respectively:

```
ClassName<T>{ ... }
...
ClassName<int> obj1 = new ClassName<int>;
ClassName<long> obj2 = new ClassName<long>;
```

The polymorphic type parameter `T` and the type arguments `int` and `long` are enclosed by the angle brackets $\langle$ and $\rangle$.

One of the main features of generics is the fact that operators $+, -$ etc. on generic type parameters can not be used by default. The evaluation of expressions that contain type parameters must be handled by evaluators. The primary task of vector objects is computation, so when designing the parameterized `Vector<T>` classes, we need to take into account evaluators.

`Vector<float>` and `Vector<double>` objects will be initialized from the polymorphic `Vector<T>` class together with vector evaluator objects of type `EvaluatorF` and `EvaluatorD`, respectively:

```
EvaluatorF eF = new EvaluatorF();
Vector<float> xF = new Vector<float>(eF, n);
float s = xF.Sum;
Vector<double> xD = new Vector<double>(new EvaluatorD(), n);
```

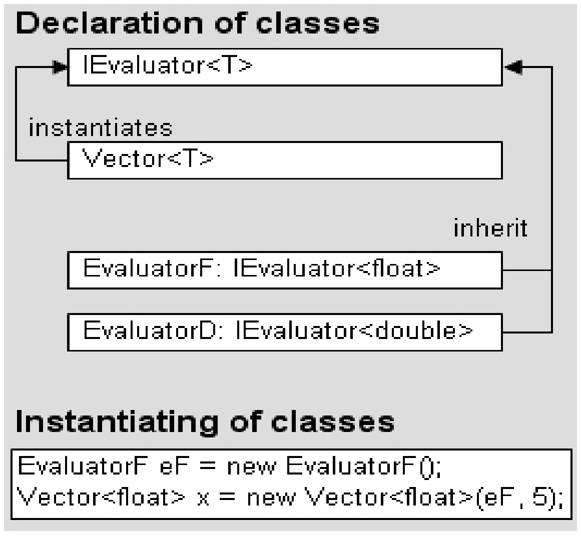
Before implementing the parameterized `Vector<T>` class, we define a polymorphic evaluator interface:

```
public interface IEvaluator<T>{...}
```

with the desired functions. These functions will be implemented by the evaluators `EvaluatorF` and `EvaluatorD` that are derived from the parametric interface `IEvaluator<T>`:

```
public class EvaluatorF: IEvaluator<float>{...}
```

The `Vector<T>` and `MathVec<T>` classes will operate with an object of type `IEvaluator<T>` enabling referencing the functions declared in interface `IEvaluator<T>`. Figure shows the architecture and use of the polymorphic vector class.



```
EvaluatorF eF = new EvaluatorF();
Vector<float> x = new Vector<float>(eF, 5);
```

Generic defining and referencing code scheme

2. POLYMORPHIC CLASS VECTOR

We introduce in this section a user-defined parametric class `Vector<T>` with three private fields and several constructors. The functionality of the class will be enhanced in the successive sections.

```
using System;

public interface IEvaluator<T>
{
      T Add(T a, T b);
      bool Equals(T a, T b);
      T Sin(T x);
}

public class Vector<T>
{
      public IEvaluator<T> _evalT;
      T[] _vecT; string _vecTName = "Vector";

      public Vector(){ }
      public Vector(int n)
      {
        _vecT = new T[n];
        _vecTName = "vecTn";
      }

      public Vector(IEvaluator<T>
        eval, int n)
      {
        _evalT = eval;
        _vecT = new T[n];
        _vecTName = "vecTn";
      }

      public Vector(IEvaluator<T>
        eval, int n, T from, T by)
      {
        _evalT = eval;
        _vecT = new T[n];
        _vecTName = "vecTn";
        T s = from;
        _vecT[0] = from;
        for(inti = 1; i<_vecT.Length; i++)
        {
            s = _evalT.Add(s, by);
            _vecT[i] = s;
        }
      }
}
```

As it was explained in the previous section, the field `_evalT` is used for calculation with parametric type values. Let us see how you can create instances of the class `Vector<T>`. If you intend to work with `float` and `double` vectors, you must first create the corresponding evaluators derived from the parametric interface `IEvaluator<T>`

```
public class EvaluatorF: IEvaluator<float>
{
  public float Add(float a, float b)
  { return a + b; }

  public bool Equals(float a, float b)
  { return a == b; }

  public float Sin(float x)
  { return (float)Math.Sin(x); }
}

public class EvaluatorD: IEvaluator<double>
{
  public double Add(double a, double b)
  { return a + b; }

  public bool Equals(double a, double b)
  { return a == b; }

  public double Sin(double x)
  { return Math.Sin(x); }
}
```

Now we are ready to create objects of types `Vector<float>` and `Vector<double>`.

```
class VectorTest
{
  static void Main()
  {
    EvaluatorF eF = new EvaluatorF();
    Vector<float> v = new Vector<float>(eF, 5, 0f, 0.1f);
  }
}
```

The created vector contains elements
(0, 0.1, 0.2, 0.3, 0.4).

3. USEFUL PROPERTIES

This section contains implementation of three simple properties for the `Vector<T>` class: `Name, Length` and `Sum`.

```
public string Name
{
  get{ return _vecTName;}
  set{ _vecTName = value;}
}

public int Length
{
  get{ return _vecT.Length;}
}

public T Sum
{
  get
  {
       T s = _vecT[0];
       for (int i=1; i<_vecT.Length; i++)
       s = _evalT.Add(s, _vecT[i]);
       return s;
  }
}
```

Their use is illustrated by the codelines:

```
Vector<float> w;
w = new Vector<float>(eF, 5, 1f, 2f);
float s = w.Sum;
Console.WriteLine("Sum = " + s.ToString());
```

The property `Name` will be used by the method `Print()` in the following section.

4. PRINT AND ACCESS TO ELEMENTS

Now we show how to print the whole vector and declare indexer for the class `Vector<T>` to provide array-like access to the elements of its instances.

First, we create a method `Print()` based on `ToString()`. The method `ToString()` returns a string that represents the current `Object`. To actualize this method for our class `Vector<T>`, we must override it.

```
public override string ToString()
{
  string s = null;
  s = _vecTName + " of length " + _vecT.Length + ":";
  for(int i = 0; i < _vecT.Length; i++)
      s += "\r\n" + _vecT[i].ToString();
  return s;
}
public void Print()
```

```
{
Console.WriteLine(ToString());
}
```

We will implement an idexer to enable access to the elements of vector x. An indexer is a member that enables an object to be indexed in the same way as an array. Indexers are similar to properties except for some differences, e.g. an indexer must have at least one parameter.

```
public T this[int nIndex]
{
  get{ return _vecT[nIndex]; }
  set{ _vecT[nIndex] = value;}
}
```

`this` stands for the element introduced by the indexer. Now you can access the vector elements through indices. To test indexer and the `Print()` method write

```
w.Name = "w";
w[0] = 1234.567f;
w.Print();
```

5. OPERATOR OVERLOADING

This section shows how we can define user-defined operators via operator overloading. To overload the addition operator you define a function called `operator +`. The `Vector<T>` addition operator + is implemented only for the case, when both of the operands are of type `Vector<T>`. The compiler distinguishes between the different meanings of an operator by examining the types of its operands.

All unary and binary operators have predefined implementations that are automatically available in any expression. User-defined operator implementations always take precedence of predefined operator implementations and their declarations always require at least one of the parameters be of the class or struct type that contains the operator declaration.

The unary operators that can be overloaded are

```
+, -, !, ~, ++, --, true, false
```

The binary operators that can be overloaded are

```
+, -, *, /, %, &, |, ^, <<, >>
```

The binary comparison operators

```
==, !=, <, >, <=, >=
```

must be overloaded in pairs:

```
== together with !=, < together with > (not >=)
```

You can not overload the other operators. So the assignment operators = and += cannot be overloaded.

```
public static Vector<T> operator +(Vector<T> x, Vector<T> y)
{
  return x.Add(y);
}

public Vector<T> Add(Vector<T> y)
{
  int m = _vecT.Length;
  Vector<T> z = new Vector<T>(m);
  for(int i=0; i<m; i++)
    z[i] = _evalT.Add(_vecT[i], y[i]);
  return z;
}
```

To test the operator +, write the code:

```
Vector<float> x = new Vector<float>(eF, 3, 0f, 1f);
Vector<float> y = new Vector<float>(eF, 3, 10f, 1f);
Vector<float> z = x + y; // <=> z = x.Add(y);
z.Print();
```

The overloading of addition operator was simple. The overloading of == operator needs a little more work. Operator == requires a matching operator != to be also defined. In addition to the implementation of the user-defined operators == and != we must override `Object.Equals(object o)` and `Object.GetHashCode()` too.

`GetHashCode()` uses the binary ^ operator that computes the bitwise exclusive-OR of its integral type operands:

```
public override int GetHashCode()
{
  int M = this.Length;
  int hc = _vecT[0].GetHashCode();
  for(int m = 1; m < M; m++)
    hc=hc ^ _vecT[m].GetHashCode();
  return hc;
}

public override bool Equals(object obj)
{
  Vector<T> x = (Vector<T>) obj;
  int m = _vecT.Length;
  for(int i = 0; i < m; i++)
```

```
    if(!_evalT.Equals(_vecT[i], x[i]))
      return false;
  return true;
}

public bool Equals(Vector<T> vT)
{
  int m = Length;
  int n = vT.Length;
  if(m != n)
    throw new Exception("Error");
  for(int i = 0; i < m; i++)
    if(!_evalT.Equals(_vecT[i], vT[i]))
      return false;
  return true;

public static bool operator ==(Vector<T> x, Vector<T> y)
{
  return x.Equals(y);
}

public static bool operator !=(Vector<T> x, Vector<T> y)
{
  return !x.Equals(y);
}
```

After executing the following test code you will see that it determines the equality of two vectors correctly:

```
Vector<float> u1 = new Vector<float>(eF, 5, 0f, 1f);
Vector<float> u2 = new Vector<float>(eF, 5, 0f, 1f);
u1.Print();
bool b = (u1 == u2); // <=> b = u1.Equals(u2);
Console.WriteLine(b.ToString());
```

6. CASTING

It would be desirable that the `Vector<T>` class implement casting to arrays. The following code enables casting between vector objects and arrays of the same type.

```
public static implicit operator T[](Vector<T> xV)
{
  int m = xV.Length;
  T[] xA = new T[m];
  for(int i = 0; i < m; i++)
    xA[i] = xV[i];
  return xA;
}
```

```
public static implicit operator Vector<T>(T[] xA)
{
  int m = xA.Length;
  Vector<T> xV = new Vector<T>(m);
  for(int i = 0; i < m; i++)
    xV[i] = xA[i];
  return xV;
}
```

The following codelines illustrate the implemented implicit casting:

```
float[] xArr = {\{}1f, 2f, 3f{\}};
Vector<float> xVec;
xVec = xArr; // from array to vector
xArr = xVec; // from vector to array
```

The presented simple casting does not solve the question of type evaluation. Casting between `Vector<float>` and `Vector<double>` with Gyro is a real challenge — try to implement it. There are several solutions that differ in complexity and performance; however, their presentation is beyond the scope of this paper.

7. MATHEMATICAL OPERATIONS WITH VECTORS

You may need a number of functions when you are building scalar expressions. The .NET Framework provides a wide range of mathematical functions that you can use when performing calculations. Besides operators for building vector expressions you may also need mathematical functions. This section describes how to achieve calculation of functions on vectors without looping.

The `System` namespace's `Math` class provides constants and static methods for trigonometric, logarithmic, and other common mathematical functions. Class `Math` can be used this way:

```
double a = Math.Sin(Math.PI/2);
```

After creating a new polymorphic public class `MathVec<T>` with a `IEvaluator<T>` private variable

```
public class MathVec<T>
{

  public IEvaluator<T> _evalT;
  public MathVec(IEvaluator<T> eval)
  {
    _evalT = eval;
  }

  public Vector<T> Sin(Vector<T> x)
  {
    int m = x.Length;
    for(int i = 0; i < m; i++)
      x[i] = _evalT.Sin(x[i]);
    return x;
  }
}
```

you may write code:

```
EvaluatorD eD = new EvaluatorD();
Vector<double> xx = new Vector<double>(eD,5,0.0,0.1);
xx.Print();
xx[0] = Math.PI/2.0;
MathVec<double> m = new MathVec<double>(eD);
Vector<double> yy = m.Sin(xx);
yy.Print();
```

Mention must be made that the implementation and use of `MathVec` differs from `Math. Math` has static methods, whereas `MathVec`, object methods, since static methods can not be built in generics so far.

8. PERFORMANCE MEASURING

Our investigation in comparing the performance of generic and non-generic vector classes confirms the known fact that, unfortunately, there is a loss of performance in computation with generic vector objects. E. Gunnerson [8] used classes for value evaluators and found a 50% loss. We observed a less than 5% loss due to using interface evaluators. You find the code for the performance comparison in the downloadable file. The question is: Whether an even better performance can be reached and how?

CONCLUSIONS

The paper presented the design and implementation of a generic vector class `Vector<T>` with some constructors and base functionality, including operator overloading, casting and support of mathematical functions with vector arguments. The key role in generic classes supporting computation over their objects plays the type evaluator. Our interface based evaluator wins over class evaluator, however, loses against the non-generic class implementation. To answer the questions whether an even better performance can be reached and how, we should wait for the new, the second, version of the MS Visual C#. We also hope that the generics in C# 2 will provide more support for handling casting.

Acknowledgements. Our thanks to VEGA for supporting the project ***1/1006/04***.

REFERENCES

1. http://research.microsoft.com/projects/clrgen/
2. http://msdn.microsoft.com/net/sscli
3. *Török Cs.* Vectorization and operator overloading in C# // Proc. of 7th Intern. Scientific Conf. «Applied Mathematics», Košice, 2002.
4. *Török Cs.* Visualization and Data Analysis in the MS .NET Framework. JINR Preprint E10-2004-136. Dubna, 2004.
5. http://www.activestate.com/Products/Download/Download.plex?id=ActivePerl
6. *Kennedy A., Syme D.* Design and Implementation of Generics for the .NET Common Language Runtime. http://research.microsoft.com/projects/clrgen/generics.pdf
7. *Lowy J.* An Introduction to C# Generics. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/csharp_generics.asp
8. *Gunnerson E.* Generics Algorithms. http://weblogs.asp.net/ericgu/archive/ 2003/11/14/52852.aspx

Received on August 31, 2004.

Редактор *Н. С. Скокова*

Подписано в печать 30.09.2004.
Формат 60 × 90/16. Бумага офсетная. Печать офсетная.
Усл. печ. л. 0,87. Уч.-изд. л. 1,28. Тираж 300 экз. Заказ № 54617.

Издательский отдел Объединенного института ядерных исследований
141980, г. Дубна, Московская обл., ул. Жолио-Кюри, 6.
E-mail: publish@pds.jinr.ru
www.jinr.ru/publish/