A framework for creating 'real engineering' calculators

Sergey L. Gladkiy

Introduction

Many engineering applications require a mechanism for formulae input. There are a lot of libraries, source code or not, which implement such possibility. They are all build with different technologies and have different features implemented. The libraries are frequently called 'math parser', 'expression parser', 'calculator' and so on. They commonly allow calculate formula, containing real numbers, real variables, functions of real arguments, algebraic operators. Sometimes logical operations and complex numbers allowed. These libraries are intended to create 'simple engineering' calculators which will evaluate complicated formula at one button click.

But real engineering applications require much more advanced features, like evaluating formula for complicated internal types (numerical modeling results, signal processing). This article is an introduction to the symbolic framework **ANALYTICS** (http://sergey-l-gladkiy.narod.ru/index/analytics/0-13) which allows easily creating 'real engineering' calculators for advanced mathematical and physical applications.

Background

Working as a programmer for creating different engineering applications, one common task appeared: create a calculator for processing internal data. This task arises for almost all serious engineering programs. Consider, for an example, the finite element modeling programs, like **ANSYS**, **NASTRAN**, **Elmer**. They all have a mechanisms of formula parsing for processing input data or result postprocessing. The need of result postprocessing arises from the fact that many users require nonstandard processing formula, which cannot be built into the application. Another example is the signal processing software, like borehole analysis programs: **WellCAD**, **Sonata**. These programs operate with complicated signal data (1D, 2D and even 3D) and contain mechanism for data processing with user defined formulae – so called 'calculators'.

All the calculators in the engineering applications work with the same principle: the user input a formula, containing references to the internal application data (borehole data, finite element results) and gets the result, which is, in common case, also the data of an internal type. The difference between the calculators is in the **data types** they operate on. Most of the mentioned 'parsers' and 'calculators', which can be found in the Internet, are not able to deal with the arbitrary data (internal application data), because they intended to process only 'built-in' types (commonly – real numbers).

Thus, for creating 'real engineering' calculators for advanced physical modeling and data processing applications, a symbolic framework required, which supports processing data of arbitrary, application specific types. The framework must satisfy (at least) the following requirements:

- Parsing complicated formula, abstracting from the data types.
- Evaluating complicated formula for arbitrary data types (including data compatibility control).
- Allowing external extensions (without modifying core framework algorithms).
- Minimal code writing for integrating in the target engineering application.

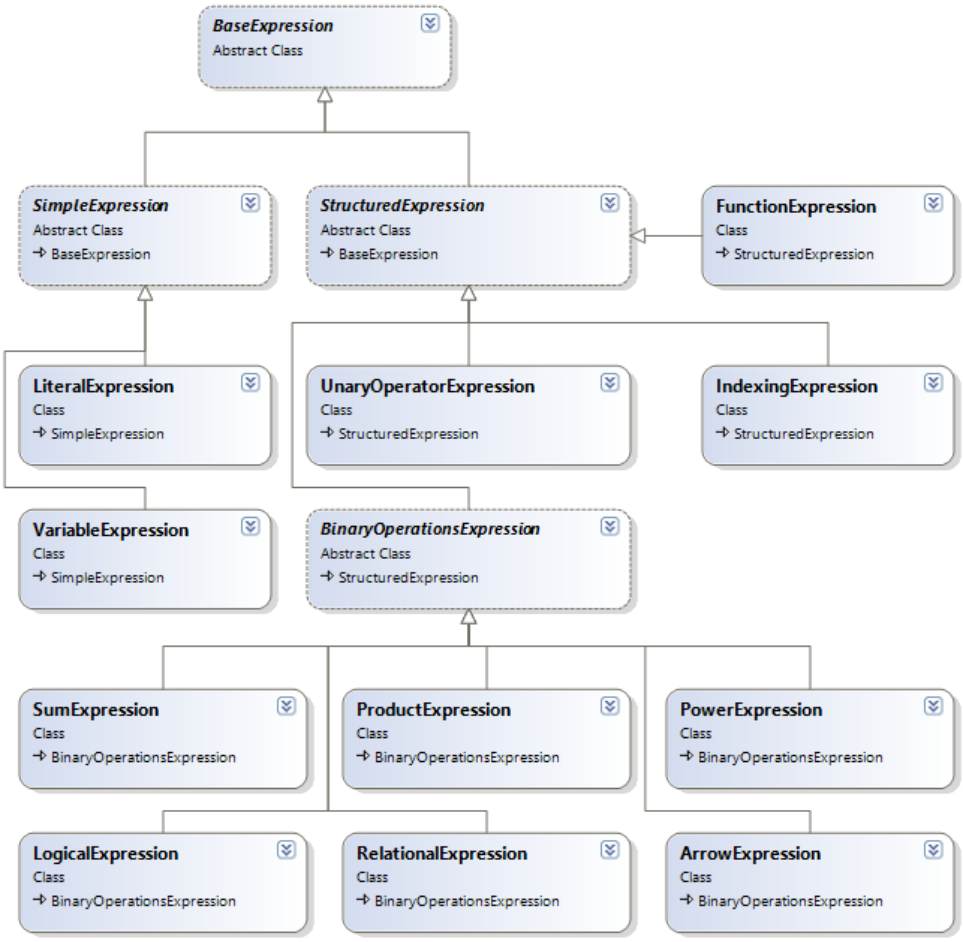
To satisfy these requirements, the symbolic framework should have strongly structured, abstract class hierarchy, which is close to math concepts, such as function, operator and others.

ANALYTICS framework

The **ANALYTICS** library is a symbolic framework, totally written in C# (even Java and Delphi versions available). The main purpose of the framework is allow programmers easily creating 'engineering calculators' for their math, physics and engineering applications, without inventing own parser for each application. The library also includes many ready-to-use extensions for working with frequently used data: real and complex numbers, common fractions, real and complex arrays and matrices, units of measurement and physical values.

For satisfying the requirements, stated above, the **ANALYTICS** framework core contains specially designed classes for modeling various math concepts. The total class hierarchy is rather complicated and is out of the article's scope. Let us consider, as examples, just some of the base classes (more information can be found in the developer guide http://sergey-l-gladkiy.narod.ru/ana_docs/ANALYTICS_C_manual.en.pdf).

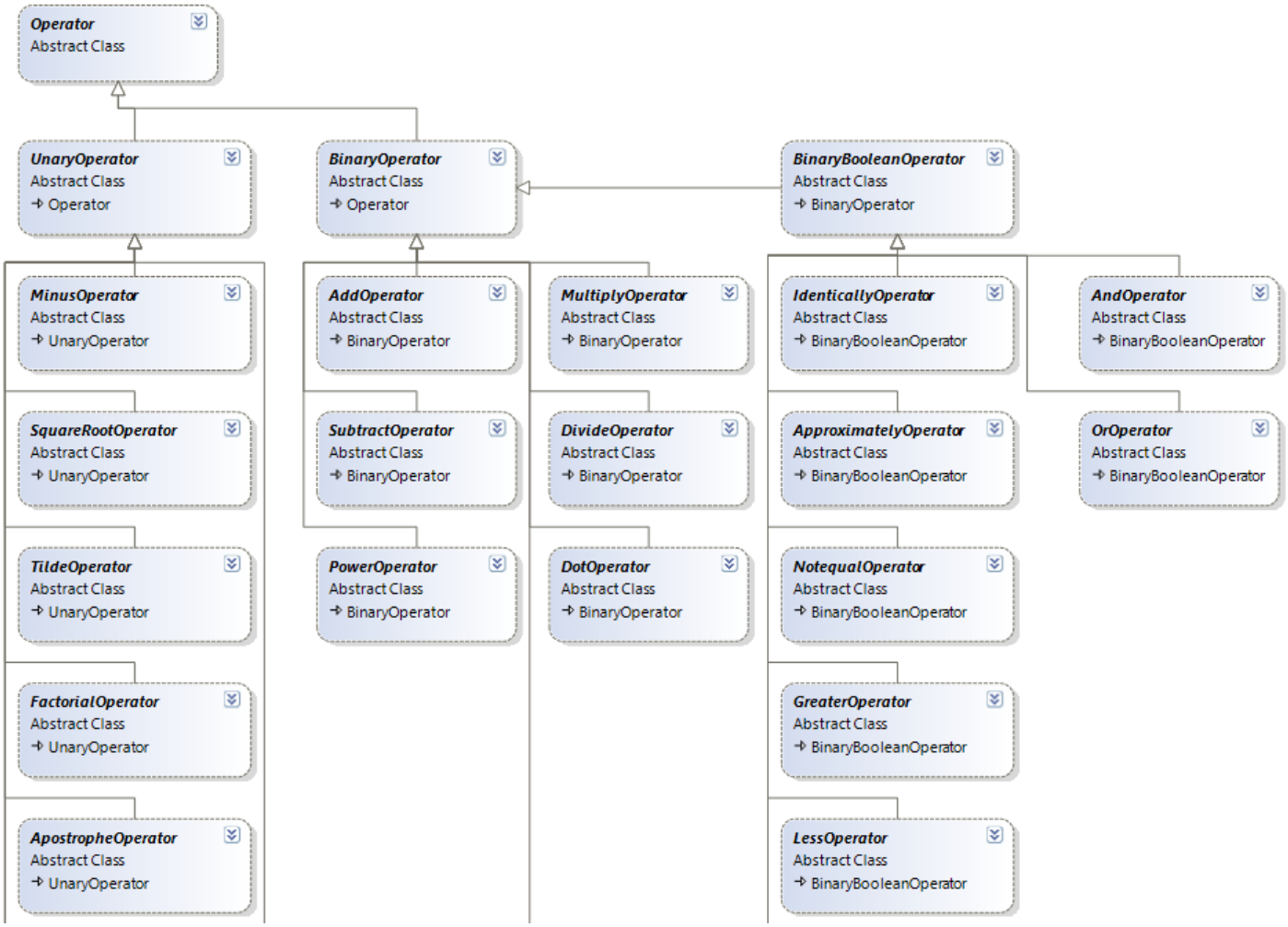
One of the base concepts is **math expression** – is a sequence of elements for which its result value can be calculated. Generally, expression contains such elements as **constants** (**literals**), **variables**, **operators**, **functions**. The **ANALYTICS** library uses the concept of the expression for presenting parsed data inside the core algorithms. It allows excluding dependency of the data types and working with pure abstract data. The expression class hierarchy is the following:



All expressions divided into two categories: simple expression – does not contain other expressions (as constant or variable); structured expression – contains other expressions as internal elements (for an example, function expression contains expressions for arguments). Indexing expression represents indexed data, like array '**A[i]**' or matrix '**M[i][j]**'. Unary operator expression is for prefix or postfix operators, like minus '**-x**' and factorial '**n!**'. Binary expression represents a sequence of operations, like additions '**x-y+1**' or multiplication '**2*x/y**'. Function expression

is for functions, like **sin(x)**. Thus, this class hierarchy allows to work with math expression concept inside the core algorithms (check the expression syntax, simplify the expression, find the derivative) in abstract manner, without referencing concrete types.

For the stage of evaluation (calculate the value of an expression), in opposite, we need to select the evaluation algorithm for every expression, which is dependent on the types of data. For this purpose, the abstractions for calculation of every expression introduced. As an example of the abstraction, there is the operator class hierarchy diagram (not total).



As can be seen from the diagram, every operator is an abstract class. It allows overriding operations (evaluation algorithms) for any data types by introducing a descendant of one of the abstract classes. The **ANALYTICS** core system then automatically finds these classes and uses them to evaluate expressions with according data type values. Such scheme also allows automatically check the compatibility of the operations, because the operator classes provide the information about operand types and the result, so, following the evaluation algorithm we can ensure that all the operations get data of permissible types.

Thus, the **ANALYTICS** framework realizes:

- Symbolic expression parsing algorithm, not depending on data types.
- Symbolic expression manipulation algorithms, not depending on data types.
- Universal expression evaluation algorithm, not depending on data types.

- Universal mechanism to override operations for specific data types with automatic control of the data compatibility.

These features allow developers using the framework to build 'engineering calculators' for various math and physics applications.

Engineering calculator example

Now, let us consider an example of the 'engineering calculator' realization, using the **ANALYTICS** framework. As the internal data for our calculator we select the **Quaternion** (https://en.wikipedia.org/wiki/Quaternion). This is really complicated math object and all its features are out of the article's scope. We will just use the ready-to-use simplified implementation of the concept – **Quaternion** class from the **.NET System.Windows.Media.Media3D** namespace.

As was mentioned above, all symbolic core algorithms in the **ANALYTICS** framework are not dependent on the data types. So, for realizing the calculator for the specific data type, only evaluation operations must be provided. First, let us introduce the operation for multiplying a quaternion with a real number. For this we just must realize the descendant of the multiply operator. The code for the class is the following:

```
public sealed class RealQuaternionMultiply : GenericMultiplyOperator<double, Quaternion, Quaternion>
{
    protected override Quaternion TypedOperation(double operand1, Quaternion operand2)
    {
        return new Quaternion(operand2.X*operand1, operand2.Y*operand1, operand2.Z*operand1,
operand2.W*operand1);
    }
}
```

The class is inherited from generic multiply operator, so it overrides the multiply operation, like '**x*Q**'. The generic parameters of the class specify the types of operands (two operand, because multiply operator is a binary one) and the result type. The only method to override for the class is '**TypedOperation**' which realized the multiplication algorithm for a Quaternion and a real number. That is all for the operation realization: the class will be automatically found by the **ANALYTICS** core and used when required (it is supposed that the class implementation is in some separate assembly, not **ANALYTICS** core, and the assembly loaded into the application domain).

Analogously, other operations for the **Quaternion** objects can be introduced. For an example, the conjugate operation can be introduced as the tilde operator '**~**' (unary prefix). The code is the following:

```
public sealed class QuaternionTilde : GenericTildeOperator<Quaternion, Quaternion>
{
    protected override Quaternion TypedOperation(Quaternion operand)
    {
        Quaternion result = new Quaternion(operand.X, operand.Y, operand.Z, operand.W);
        result.Conjugate();
        return result;
    }
}
```

As the conjugate operator is unary, only one operand type specified in addition to the result type.

Another fine feature of the **ANALYTICS** framework is that sometimes it can automatically implement operations for the program's specific types – **without any additional code**! Frequently, there already exists the operator overloading for the program's specific types – for an example, the Quaternion type has operator overloadings for

addition and multiplication. So, the **ANALYTICS** core will automatically detect this overloadings and use them when required. It means, that such operations as **Quaternion+Quaternion** or **Quaternion*Quaternion** need no code writing at all.

Analogously we can introduce various functions for Quaternion arguments, like '**sin(Q)** ', '**log(Q)**' and so on. Detailed information about function introducing can be found in the manual http://sergey-l-gladkiy.narod.ru/ana_docs/ANALYTICS_C_manual.en.pdf .

Now it is time for testing our 'quaternion calculator'. We realized basic algebraic operations and the conjugate for the quaternions and then it is possible to calculate various formula, containing quaternions. For introducing the quaternion data in the evaluation process the common way is to add variables for the data. The code for this is the following:

```
_translator = new Translator();
Variable v1 = new ObjectVariable("Q1", new Quaternion(1, -1, 0, 2), typeof(Quaternion));
_translator.Add(v1);
Variable v2 = new ObjectVariable("Q2", new Quaternion(-1, 2, 1, 0), typeof(Quaternion));
_translator.Add(v2);
Console.WriteLine(v1.ToString());
Console.WriteLine(v2.ToString());
```

First, we created the **Translator** instance – it is the main class for manipulating symbolic expressions in the **ANALYTICS** framework. Then we created the variable '**v1**' with name '**Q1**' and value Quaternion (1 -1 0 2). The **ObjectVariable** class is intended for holding data of any type – Quaternion in this case. We added the variable to the translator instance. The same was made for another variable '**Q2**'. The console output is:

```
Q1 (Quaternion) = 1;-1;0;2
Q2 (Quaternion) = -1;2;1;0
```

Now we are able to evaluate expressions containing variables '**Q1**' and '**Q2**'. For an example:

```
string f = "~Q1-2*Q2";
Object r = _translator.Calculate(f);
Console.WriteLine(f+" = "+r.ToString());
```

which gives the following console output:

```
~Q1-2*Q2 = 1;-3;-2;2
```

It should be noted, that only operations for '~' and '*****' operators were introduced in the system explicitly (creating special classes). The operation for the '**-**' operator was automatically recognized by the **ANALYTICS** core from the overloaded subtract operator of the Quaternion structure.

But, trying to evaluate the following formula:

```
string f = "Q1+Q2!";
Object r = _translator.Calculate(f);
Console.WriteLine(f + " = " + r.ToString());
```

the exception occurred

```
Unary operator '!' not found for operand type 'Quaternion'.
```

This is because we have not defined the factorial '**!**' operator for the Quaternion type and no overloaded operator was found for the structure. So, the system automatically controls the type compatibility for any operation in the formula.

Conclusions

In the article some base features of the **ANALYTICS** library described. The framework allows easily creating 'engineering' calculators for different advanced modeling applications. The framework has specially designed class hierarchy for integrating it into any program with minimal code and without modifying the core algorithms. The flexibility of the framework confirmed by the fact, that it was successfully integrated into many numerical libraries, such as **NMath** (http://sergey-l-gladkiy.narod.ru/index/nmath-analytics/0-21), **ILNumerics** (http://sergey-l-gladkiy.narod.ru/index/il_analytics/0-18), **Extreme Optimization** (http://sergey-l-gladkiy.narod.ru/index/extreme-analytics/0-22).

Many other unique features of the framework are out of the article's scope. They can be tested on the example applications available: an example of engineering calculator for FEM results postprocessing http://sergey-l-gladkiy.narod.ru/ana_download/FEMPOSTPRO.DEMO.1.0.Setup.rar; ready-to-use borehole data processing system http://sergey-l-gladkiy.narod.ru/index/wld_cad/0-17 realized as the engineering calculator with the **ANALYTICS** framework.