Symbolic computations in .NET Framework

Igor Medvedev, Yuri Okulovsky

Ural Federal University
Yekaterinburg, Lenina str. 51
Email: yuri.okulovsky@gmail.com

***Abstract*—The computer algebra system (CAS) is a software that is used for various symbolic computations like simplification and differentiation. CAS are based on the transformation rules that rearrange expressions according to the mathematical laws. We consider development of CAS in the .NET Framework. Currently, there is only one .NET software product with some features of symbolic computations, and no full-fledged implementation of the transformation rules in .NET. In the same time, the .NET framework provides many features for innovative techniques of rules' development, and therefore a .NET solution for the transformation rules can offer the new approaches to the computer algebra systems. In this paper, we describe these techniques and their implementation.**

***Index Terms*—symbolic computations, computer algebra systems, transformation rules, .NET framework**

I. Introduction

The computer algebra system is a software that performs symbolic computations. Typical examples of such computations are simplification of an expression into a smaller one, operations like differentitation and integration, logical interference and so on [1]. The computer algebra systems (CAS) are widely used in mathematics and computer sciences. In CAS, mathematical structures are represented in the symbolic form. It differs CAS from numerical analysis systems, which manipulate numbers. The most typical data representation in CAS is the syntax tree, an example of which is shown in Figure 1. Simplification, differentiation and other symbolic computations are performed as sequences of the elementary transformation rules, each rule rearranges a tree. The example of such transformations is given in Figure 2.

For the standard tasks, like simplification or calculus operations, many CAS systems are developed. In this area, the market has many solutions, including big enterprise packages like Mathematica, Maple or Mathlab, and small open-source projects. However, sometimes we need a computer algebra system not only to make a reasearch with its aid, but also to study CAS itself. For example, in [4] we propose a new genetic programming algorithm that combines simplification and induction as the uniformed parts of the evolutionary computations. To do so, we implemented the new transformation rules for mutation and crossover, and used them together with the simplification rules in the evolutionary algorithm. With the existing CAS, we would need the access to the system's core structures, because new rules should be programmed and merged with the existing ones, and then used in the completely new algorithm. For research of the transformation rules and computer algebra we need a different kind of CAS. The graphical user interface and the amount of the supported algorithms are not so important, while the easy and understandable access to the core structures is.

In this article, we present our approach to the transformational rules and the computer algebra algorithms. The most prominent distinction from the existing solutions is implementation in the C# language and the .NET framework [5]. The .NET Framework is a modern developing tool, widely used in science and education as well as in the commercial software development. .NET offers many convenient features, and some of them, like expression trees and lambda expressions, seem to be especially useful for the computer algebra. However, most of the existing CAS are programmed in old, maybe even

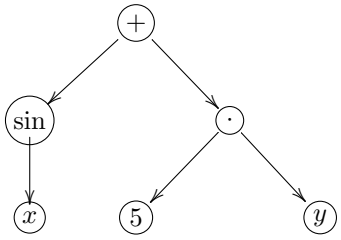


Fig. 1. Example of syntax tree for function $f(x, y) = \sin x + 5y$. Here x and y denote variables, 5 - a constant, and other nodes are operations of addition, multiplication and sine function.

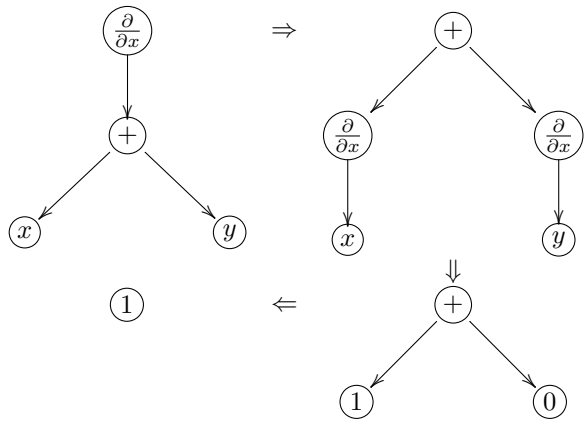


Fig. 2. A chain of transformations that computes $\frac{\partial(x+y)}{\partial x}$.

obsolete, languages like C, LISP or C++. We believe that the usage of the modern programming techniques for the computer algebra gives a fresh look on the symbolic computations, and could be resulted in the new approaches to computer algebra systems.

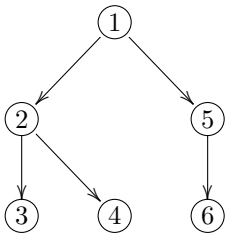
To our knowledge, only one .NET solution exists, named Math.NET [2]. However, it could hardly be considered as a full-fledged computer algebra system. The transformation rules are not programmed as a separate entity, and are substituted by Visitor pattern [3], that processes nodes in a tree according to its function. This decision hampers the system's expanding, because the addition of new operations demands alterations in the existing code. Moreover, even operations like differentiation of the exponential function are still not implemented. Of course, we do not need a new CAS system to perform simplification or differentiation when writing a .NET application. In most cases, we can just run the CAS application, perform all the necessary computations, and write the result back to the program. Or, we can use CAS system on the lower level and run the corresponding methods using .NET legacy code interoperation. However, seamless integration of computer algebra into .NET framework can be usable for some applications.

II. Rule definition in C#

Application of the rule can be subdivided into the following stages. In the *sampling* stage, the system that applies rules (which is called the *driver* below) selects some tree-like structure from the syntax tree, and presents its nodes as a tuple. In the *selection* stage, the driver sort out the tuples that do not meet the specified criteria. In the third stage, called *modification*, the driver transforms the tree according to the rule. In the most widespread case, the rule processes one tree. For such *unary* rules, the tree is rearranged with replacements. In some cases, the rule processes more than one tree. For example, the modus ponens rule in logical interference accepts two trees $A \to B$ and A and produces B. In this case, new tree is to be created from the selected nodes.

A. Sampling

To perform the sampling stage, we should specify the tree-like structure that we are searching for in a tree. Also, we need to map the nodes in the structure into a positions in a tuple. We used query strings of our own syntax to do that. Let us demonstrate the syntax of query string with the examples, shown in Figure 3.



A	(1)	The root of the tree
?A	(1), (2), (3), (4), (5), (6)	An arbitrary node in a tree
?A(B)	(5,6)	An arbitrary node with its unique child
?A(B,C)	(1,2,5), (2,3,4)	An arbitrary node in a tree with its two children in the fixed order
?A(.B,.C)	(1,2,5), (1,5,2), (2,3,4), (2,4,3)	An arbitrary node with its two children in the unconditioned order
?A(.B)	(1,2), (2,3), (2,4), (5,6)	An arbitrary node with its arbitrary child
?A(?B)	(1,2), (1,3), (1,4), (1,5), (1,6), (2,3), (2,4), (5,6)	An arbitrary node with its arbitrary descendant
?A(?B(?C))	(1,2,3), (1,2,4), (1,5,6)	An arbitrary node, its descendant and the descendant of the descendant
?A(?B(C,D))	(1,2,3,4)	An arbitrary node that has a descendant with two children

Fig. 3. Various examples of query strings. Queries are applied to the presented tree, its output is specified in the table.

The sampling algorithm is a depth-search algorithm that builds a correspondence between a given tree and a parse tree of a query string. Suppose the algorithm observes some node. It proceeds further according to the following rules:

- if the corresponding query substring has a form $(A_1, \ldots, A_n)$, the algorithm checks that the count of observed node's children is n, and assigns A_i to corresponding child.
- if the corresponding query substring has a form $(.A_1, \ldots, .A_n)$, the algorithm checks that the count of node's children is greater than or equals n. Then it runs through all possible assignments of A_i to observed nodes children. For all such assignments, further search will be performed.
- if the corresponding query substring has a form $(?A_1, \ldots, ?A_n)$, the algorithm assigns A_i to every possible combination of the node's descendants. For all such assignments, further search will be performed.

To encode the rule, we should specify the query string in the program. It could be done by encoding of the query as a constant string. However, it is not convenient due to the possible mistakes in the query's syntax, such as brackets mismatch or incorrect letters. To eliminate such errors, we implement query strings definition with square brackets overriding.

Consider the code in Listing 1 that specifies a query string. Here `Rules` is a static class that is purposed to create rules. Static method `Select` accepts a query and creates a `SelectClause` instance, which is used to define selection and modification, as is shown below. `AnyA`, `ChildB` and

Listing 1 Specification of sampling in the program.

```
public class  Creator : RulesCreatorBase {
  public void Create() {
    Rules.Select(AnyA[ChildB,ChildC])...
} }
```

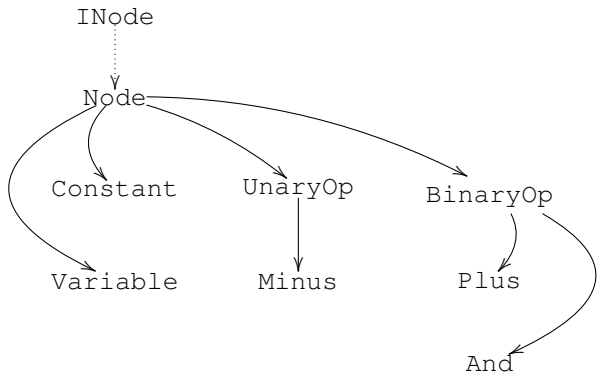


Fig. 4. A fragment of operators' type hierarcy, used to selection procedures.

`ChildC` are defined in `RulesCreatorBase` as properies that corresponds to $?A$, $.B$ and $.C$ elements of query strings.

B. Selection

The selection stage can be further subdivided into the type selection and the custom selection. The type selection checks the types of nodes in the selected tuple and rejects the tuple in the case of mismatch. The custom selection can check additional conditions, e.g. a value of a constant. The type selection must be performed prior to the custom selection, because the appliability of the custom selection depends on the node's type. For example, to check that the constant's value is zero, we have to be sure that the node corresponds to a constant, not to an operator.

The operations in our solution follows the type hierarchy that is shown in Figure 4. Each operation type has also its generic-analog that specifies the type of returning value. For example, `INode<T>` inherits `INode` and is implemented by `Plus<T>` that inherits `Plus`. Hence we can select operations either by their function (`Plus`), their returning type (`INode<T>`), or the combination of these properties (`Plus<T>`).

When programming selection, a challenge emerges of how to subject the tuple to the selection's condition. We cannot store selected nodes in the array or the list structures, because they do not allow specifying different types for elements. With the array representation, selection could only be performed in the following way:

```
array =>
    array[0] is Plus &&
    array[1] is Constant &&
    (array[1] as Constant).Value==0
```

Of course, constant casting and addressing is a potential cause of the type errors. We have developed an elegant solution for the selection with the aid of generic methods and code generation. Consider the code in Listing II-B. Here `SelectClause` is a class, which instances are created by `Rules.Select(...)` method. We can then call the `Where` method as shown inside the `Main` method. Note how naturally and easy-to-write this rule's definition is in comparison with casting of array elements.

Listing 2 Code pattern for selection stage, given on the example of three arguments

```
class WhereArg<T1,T2,T3> {
  public T1 A; public T2 B; public T3 C;}

class WhereClause<T1,T2,T3> {

Func<WhereArg<T1,T2,T3>,bool> selector;

public bool Where(object[] nodes) {
if (!(nodes[0] is T1)) return false;
if (!(nodes[1] is T2)) return false;
if (!(nodes[2] is T3)) return false;
var arg=new WhereArg<T1,T2,T3> {
   A=(T1)nodes[0],
   B=(T2)nodes[1],
   C=(T3)nodes[2] };
return selector(arg);
} }

public class SelectClause {
public static WhereClause<T1,T2,T3>
  Where<T1,T2,T3>
       (Func<SelectionArg2<T1,T2,T3>,bool)
  {...}
}

public class Creator : RulesCreatorBase {
public void Create() {
Rules.Select(A[B,C])
.Where<Plus,Constant,INode>
     (z=>z.B.Value==0);
} }
```

When we specify generic-arguments of `Where` method, we define the type selection that should be performed. If the type is not important, we just specify `INode`, since it is a basic interface for all nodes. Setting `Plus` as a type for the first element of the selected tuple allows us to specify a desired operation. Settings `Constant` as a type of the second element throws away all the tuples where the second element is not constant. Also, we may specify the custom selection condition `z.B.Value==0`, because the type of the second node is known to compiler after the type selection. Therefore, cast errors are catched on a compile stage. In addition, we can use the same letters for elements in selection that we have used

in sampling.

Declarations of `WhereArg` and `WhereClause` classes as in Listing II-B must be done for all different count of the arguments. We have used a code generation technique to produce declarations for up to 20 arguments, which is believed to be enough for our purposes.

Listing 3 Code pattern for modification stage, given on the example of three arguments.

```
class NodeDecorator<T> {
public T Node { get; private set; }
public void Replace(INode newNode) {  }
}

public class ModInput<T1,T2,T3> {
NodeDecorator<T1> A;
NodeDecorator<T2> B;
NodeDecorator<T3> C;
}

public class WhereClause<T1,T2,T3> {
public RuleInstance
   Mod(Action<ModInput<T1,T2,T3>> action)
      {...}
}

public class Creator : RulesCreatorBase {
public void Create() {
Rule.Select(A[B,C])
.Where<Plus,Constant,INode>
     (z.B.Value==0)
.Mod(z=>z.A.Replace(z.C));
} }
```

C. Modification

When the selection stage is over, we obtain several tuples that could be modified in the modification stage. However, only one of them will be actually processed, because application of the rule may invalidate other tuples. Therefore, the modification stage processes only one of the selected tuples. We have developed a "clean" modification, which does not affect the initial trees. Instead, in the modification stage we create a copy of the selected trees, and perform modification on the copy. To do that, we must find the roots of the nodes, presented in a given tuple, clone them, and further process a newly created tuple with a corresponding clones of the nodes.

For unary rules, modification turns into one or several replacements, which replace one of nodes with another. The method for replacement could not be placed in the `INode` interface, since it would give to the user an access to insecure replacements of the node. Therefore, we create a decorator class that wraps each node, and `ModInput` generic class that contains several decorators, as shown in Listing 3.

As we see, generated `WhereClause` class contains Mod method that processes a given typified tuple of decorators. Inside the given action, we have access both to the typified node (and therefore to the values and other type-specific content of the node) and to the Replace method. We can now declare a rule as in the Main method.

In case of not unary rules, we include the Produce method into WhereClause with the same approach: accept lambda that transforms ModInput into a node, and this node is considered as the root of the resulting tree.

III. Conclusion

The concepts above were successfully implemented and tested on various rules, mainly for simplification and differentiation. The computer algebra library for the .NET framework was written, with the following features:

- conversion of the .NET lambda expressions into the operation trees;
- simplification and differentiation of the .NET lambda expressions;
- linear regression of the .NET lambda expressions.

The developed rules were also successfully used in the genetic programming experiments, described in [4].

Acknowledgments.

The work is supported by the program of President of Russian Federation MK-844.2011.1.

References

[1] J. Grabmeier, E. Kaltofen and V. Wiespfenning. *Computer algebra handbook* Springer-Verlag, 2003

[2] Math.NET Project. http://www.mathdotnet.com/

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley, 1994

[4] Ya. Borcheninov and Yu. Okulovsky *Genetic programming with embedded features of symbolic computations*, In Proceedings of international conference of Knowledge Discovery and Information Retrieval, 2011.

[5] A. Troelsen *Pro C# 2010 and the .NET 4 Platform* Apress, 2010