A new algorithm of symbolic expression simplification based on right-threaded binary tree

Yang Dechuan[1,a], Song Zeliang[2,b], Yu Ri[3,c], Jiang Xihe[4,d] ,Zhao Hongwei[5,*]

[1]College of Computer Science and Technology, Jilin University, Changchun，130012,China

[2]College of Computer Science and Technology, Jilin University, Changchun，130012,China

[3]College of Computer Science and Technology, Jilin University, Changchun，130012,China

[4]College of Computer Science and Technology, Jilin University, Changchun，130012,China

[5]College of Computer Science and Technology, Jilin University, Changchun，130012,China

[a]1209350192@qq.com, [b]songzeliang@163.com, [c]649555717@qq.com,[d]1468751157@qq.com，*zhaohw@jlu.edu.cn

*Corresponding author

Keywords: Right-thread Binary Tree; Postfix Expression; Symbolic Expression Simplification

Abstract. Considering that it's difficult to dealing with symbolic expressions which are stored in the form of strings, we put forward an idea that the symbolic expression is stored in a right-thread binary tree, and we minimize the right-thread binary tree so that the symbolic expression could be simplified. For a right-threaded binary tree that represents a symbolic expression, we traverse its nodes in order of its corresponding postfix expression. During the traversal, we simplify the expression. The Algorithm not only minimizes the tree, but also transforms the tree into an expression string. Experiments show that most symbolic expressions can be simplified by the method.

As we all know, expression simplification is one of the most fundamental functions for any symbolic computation software and it's also the core of the software[1]. There are many studies and algorithms on symbolic expression simplification in the past. However, no general algorithm has been found that could simplify a symbolic expression perfectly. So, both algorithm and innovative research for data structure about symbolic expression simplification are of enormous significance.

Taking various functions models into account, this text makes use of people's thinking pattern and skills, and simplify the symbolic expression by minimize the right-thread binary tree during LDR traversal. What we get by traversing the right-thread binary tree in order of LDR is the symbolic expression's postfix expression (also called Reverse Polish Expression). When dealing with a symbolic expression in the form of postfix expression, we can take no account of the priority of operators. So when it turns to an operator, we can directly take out front operands to operate. As a result, the LDR traversal of the tree makes it possible we simplify symbolic expressions. The text will explain the algorithm in detail.

1. Relevant Data Structure

What data structure we choose to represent the symbolic expression will affect the simplicity of the algorithm. As is known to all, any symbolic expression can be regarded as representations of tree structures[2], not as one- or two-dimensional configurations of symbols. For example, the symbolic expression $y = x+1+2+3*sin(x)$ has the tree representation as follows.

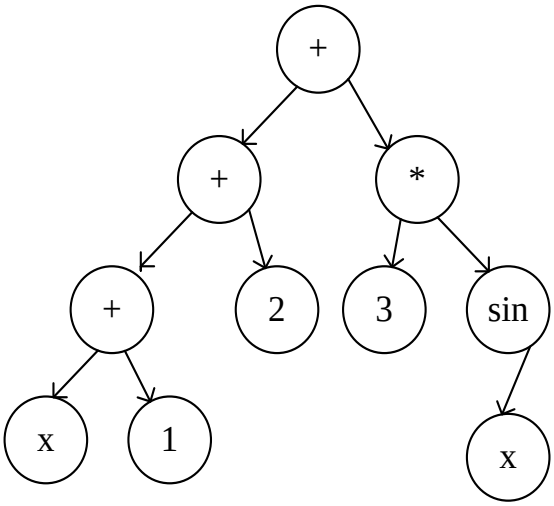

Fig.1 Symbolic expression represented by a tree

Traverse the binary tree in post order and then we get the postfix expression of the symbolic expression[3]. However, it is difficult to traverse a binary tree in post order. So it is important to note that, even though the data structure in Fig.1 is similar to a binary tree, we are treating it as a tree. There is a one-to-one correspondence between a tree and a binary tree. That is, any tree can be replaced by a binary tree and vice versa. In addition, postorder for a tree corresponds to inorder, not postorder, for binary trees, and inorder traversal of binary tree is easy[4]. So, we transform the tree in Fig.1 into its corresponding binary tree in Fig.2. In order to convenience, we add a node “Y” as the root node of the tree.

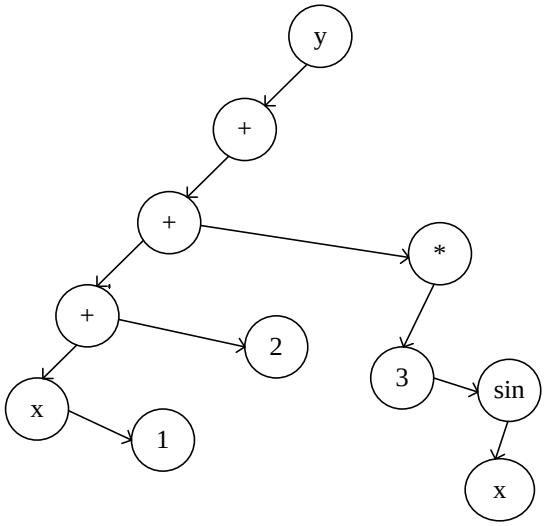

Fig.2 Symbolic expression represented by a binary tree

For a node, if there is no arrow line pointing to the next node in inorder, add an imaginary line. So we build the right-threaded binary tree in Fig.3.

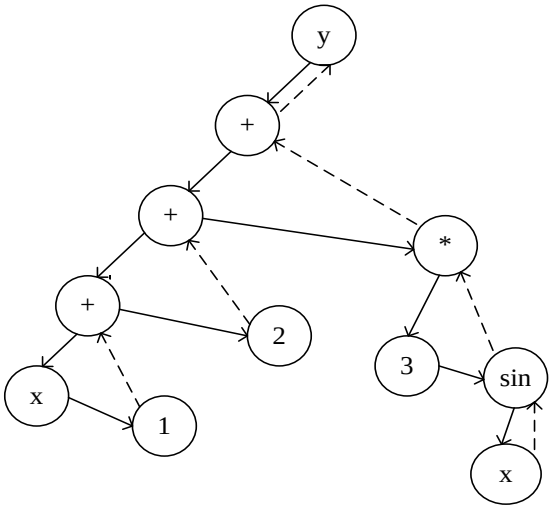

Fig.3 Symbolic expression represented by a right-threaded binary tree

The tree structure for the symbolic expression with which we will be dealing with have nodes of the following form[5]:

LCHILD	RCHILD	RTAG
TYPE	INFO	

Here RCHILD and LCHILD have the usual significance.. The TYPE field is used to distinguish different kinds of nodes: TYPE=0 means that the node represents a constant, and INFO is the value of the constant. TYPE=1 means that the node represents a variable, and INFO is the name of the variable. TYPE=2 means that the node represents an operator that has only operand. TYPE=3 means that the node represent a binary operator. And INFO is the name of the operator. Furthermore, when a node has a right child, RTAG=0. Otherwise, RTAG=1.

2. Algorithm Description

When we deal with symbolic expression evaluation, differential and integral operation, postfix expression is used frequently. And right-threaded binary tree is used regularly for its convenience to postorder traversal. So, now we assume that a symbolic expression which is not simplified has been stored in the form of right-threaded binary tree, and don't care about how to set up the tree. Then we introduce the algorithm of symbolic expression simplification.

In order to simplify the symbolic expression, we need to traverse each node of the tree in postorder. So we traverse the right-threaded binary tree in inorder. When it turns to an operator node, we execute following operations:

1. Rebuild subtrees of the node.
2. Save the symbolic expression represented by the subtree whose root node is the current node.

In order to realize the algorithm, we introduce some variables, that is P, P1, P2, Q1, Q. P is the current node and P1, P2 are respectively P's left and right child. On one hand, if P represents a binary operator, Q1 represents the symbolic expression of P1 subtree, and Q represents the symbolic expression of P2 subtree. On the other hand, if P represents a unary operator, P has only one child, that is P1. And Q represents the symbolic expression of P1 subtree.

The algorithm is described as follows:

S1. [Initialize.] Set $P \leftarrow$ first(Y) (namely, the first node of the tree in Fig.1 in postorder, which is the first node of the corresponding binary tree in Fig.3 in inorder.)

S2. [Simplify] Set $P1 \leftarrow$ LCHILD (P). If $P1 \neq \Lambda$, also set $Q1 \leftarrow$ RCHILD (P1). Then perform the function simplify (P), which will be described below.

S3. [Restore link.] If TYPE (P) ≥ 2, which means that P represents an operator, set RCHILD (P1) $\leftarrow$ P2.

S4. [Next node]. Set P2$\leftarrow$P, P$\leftarrow$next (P) (next(P) is the next node of the right-threaded tree's inorder traversal.), if RTAG(P2)=0 (P2 has a right brother), then set RCHILD(P2)$\leftarrow$Q (We temporarily destroy the structure of the right-threaded binary tree, so that a link to the expression of P2 is saved for future use. So in S3, we restore the missing link.).
S5. [Over?]. If P$\neq$Y, return to step S2. Otherwise, Q represents the result of the simplification algorithm.

Now that general steps are listed, we describe the simplify(Node P) function.
Since simplify(P) minimizes the P subtree, different P nodes are classified as follows:

(1). Constant or variable: In this situation, P has no child, and P1, P2, Q1 and Q make no sense.

$Q \leftarrow Node\left(INFO\left(P\right)\right)$, that is building a new node which represents P's content.

(2). Unitary operator (log function, trigonometric function, anti-trigonometric function):

If TYPE(P1)=0, which means that P represents a constant, calculate the value of P subtree. The value should be stored in Q and replace P's original value. And then, delete P1, which means that the subtree structure has been modified.

If TYPE(P2) $\neq$ 0, which means that P represents a variable or operator. $Q \leftarrow Node(\left(INFO\left(p\right) + INFO\left(Q\right)\right)$.

(3). Binary operator:

We take plus operator for example to introduce this situation.

Define 3 variables namely *var*, *operand* and *curnode*. If P1 and P2 are both constant, add the two constants into *operand*. If P1 or P2 is a variable, add the number of variables into *var*. If P1 or P2 represents "+" or "-", continue to visit the node's children and add or subtract their values into *var* or *operand*. If P1 or P2 presents operator that has higher priority, save the node as variable *curnode*. In the end, build a new subtree to replace P on the basis of the three variables *var, operand* and *curnode*. And then store the symbolic expression in Q.

We summarize the simplification rules and methods to complete the simplify function. So the other situations are similar to plus.

3. Conclusion

Experiments shows that the algorithm of symbolic expression simplification which is accomplished by traversing the right-threaded binary tree can simply deal with primary symbolic expression. Besides, the algorithm can be easily improved through expanding the function *simplify()* to achieve more complicated symbolic expression. Starting from people's thinking pattern and skills, the method that we call function *simplify()* to simplify each node of the expression during traversal is an effective way to simplify symbolic expression.

Acknowledgments

The corresponding author is Zhao Hongwei. The authors are grateful to the anonymous reviewers for their insightful comments which have certainly improved this paper. This work is supported by Plan for Scientific and Technology Development of Jilin Province (20140101184JC).

References

[1] J. Fu Hongguang, Zhong Xiuxin, Zeng Zhenbing. Simplifying Trigonometric Expressions Automatically by Computer. *Chinese Journal of Computer*, vol.29, 2006.
[2] M. Mark Allen Weiss. Data Structures and Algorithm Analysis in C, vol.4, pp70-73.
[3] J. Hu Yun, Mao Wannian. A New Method That Transforms Infix Expression into Postfix Expression. *Journal of Chengdu University*, vol.27, no.1, 2008.
[4] M. Knuth. The Art of Computer Programming. vol.1, pp 357-358.
[5] J. Li Lianzhi, Guo Fushun. An Algorithm of Symbolic Integration. *Journal of Harbin Institute of Technology*. Vol.2, 1983.