Design and Implementation of Improved Boolean Expression Evaluation Algorithm

Xiong Fengguang, Han Xie, Han Yan

Institute of Electronics and Computer Science and Technology, North University of China, Taiyuan, 030051, China

School of Information and Communication Engineering, North University of China, Taiyuan, 030051, China

E-mail: 4355677@qq.com

Abstract

Boolean expression evaluation algorithm is widely used to deal with process control in application systems. But, in the current Boolean expression evaluation algorithm, the name, priority and operation rule of operator are hard-coded in algorithm, which makes some serious problems that the algorithm can not expand its operator and the expansibility is poor. An improved Boolean expression evaluation algorithm is proposed in this paper. In the algorithm, the name, priority and operation rule of operator are subtracted as an interface from the original algorithm, and the new algorithm is re-designed and implemented based on the operator interface. Through the new algorithm, operation rule of operator can be invoked by operator interface, which ensures the independence of algorithm, and allows users to define new operator or update the original operation rule of operation, and the scalability of algorithm is strengthened.

Keywords: Boolean Expression Evaluation, Infix Expression, Postfix Expression

1. Introduction

Boolean functions [1][2][3] occur with great frequency in computing. There is hardly a program written without an "if clause" whose arguments must be evaluated. The arguments are Boolean expression in fact. A Boolean expression can be viewed as defining a set of contexts, namely those for which the expression evaluates to true. A Boolean expression is combined by one or more operand and Boolean operator [4][5][6]. The simplest Boolean expression contains only a literal constant or variable, and a more complex is constituted by more operands and operators. Boolean expressions have been widely used in process control of application systems, and the logical correctness of the control condition is a necessary condition for the correct implementation of application systems. For example, document retrieval, information inquiry, information distribution, workflow engines, and other applications will use Boolean expressions as conditions of its process control [7][8].

General speaking, a Boolean expression evaluation does not exist as an independent software, but as part of the application system in which a Boolean expression is evaluated as an integral part [9][10]. The advantage of this method is able to deal with Boolean operation in the application system in accordance with its algorithms and the processing efficiency is more efficient [11]. But the following problems exist: first, the logical expression evaluation code is not portable to other applications; second, when the customer needs a secondary development for the application system, he can not change the code of Boolean expression evaluation in the application system, therefore, when the application system has been deployed, he can not add a new operator to the Boolean expression evaluation algorithm, and can not set the name, priority, operation rule of the new operator, and he even can not change the priority, operation rule of the original operator which has been built in the application system. All in all, the problem of traditional Boolean expression evaluation algorithm is the poor scalability and portability.

In this paper, an improved Boolean expression evaluation algorithm is proposed. In the algorithm, the name, priority and operation rule of operator are subtracted as an interface from the Boolean expression evaluation algorithm, and Boolean expression evaluation algorithm is re-designed and implemented based on the operator interface. By the new algorithm, operation rule of operator can be invoked by operator interface, which ensures the independence of expression evaluation algorithm, and allows users to define new operator or update the original operation rule of operation, and the scalability
of the Boolean expression evaluation is strengthened.

The rest of the paper is organized as follows. Section 2 is the description of Boolean expression evaluation algorithm. Section 3 focuses on design and implementation of improved Boolean expression evaluation. Section 4 focuses on special application and experiment. Finally, we end this paper with conclusion.

2. Boolean expression evaluation algorithm

Under normal circumstances, the lexical rules and grammar rules of the Boolean expression are defined as follows:

```
<Letter>::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<Digital>::= 0|1|2|3|4|5|6|7|8|9
<Character>::= '<Letter>'
<Positive integer>::= Digital(Digital)
<Integer>::= [+/-]<Positive integer>
<Float>::= [+/-]<Positive integer>.<Positive integer>
<Boolean>::= true|false
<Literal>::=<Character>|<Integer>|<Float>|<Boolean>
<Factor>::=<Literal>|<Expression>|(< Expression >)
<Item>::= <Factor> > <Factor> | <Factor> < <Factor> | <Factor> >= <Factor> | <Factor> <= <Factor> | <Factor> == <Factor> | <Factor> != <Factor>
<Boolean Expression>::= <Item> || <Item> | <Item> && <Item>
```

The Boolean expression, which puts operator in the middle of the operand, is called infix expression, whose operator includes: ">", ">=", "<", "<=", "==", "!=", "&&", "||". Because the operator of infix expression has the priority, as well as parentheses can change the order of operations, the compiler generally does not use infix to deal with expression or calculate expression. The solution is to convert the infix Boolean expression to a postfix Boolean expression. Postfix expression can not contain parentheses, and operator is on the back of the two operands which are involved in the operation. When evaluating the postfix expression, all evaluated operations are carried out strictly from left to right according to the order of operator. Using the postfix expressions to describe a Boolean expression, the definition of lexical rules and grammar rules make the following changes:

```
<Factor>::= '<Literal>'<Expression>
<Item>::= <Factor> > <Factor> | <Factor> < <Factor> | <Factor> >= <Factor> | <Factor> <= <Factor> | <Factor> == <Factor> | <Factor> != <Factor>
[Boolean Expression]::= <Item> <Item> || | <Item> < Item> &&
```

During an infix expression is converted to a postfix expression, the more importance of change is that the influence of priority and parentheses for the order of expression evaluation is wiped off. Otherwise, in order to convert easily an infix expression to a postfix expression, a pretreatment for scanning infix expression is necessary. According to the pretreatment, the operands, operators, and parentheses are partitioned orderly from the infix expression, and are stored into a string array. Supposing that infix expression is "InfixExp", and the string array, storing pre-processed expression, is "InfixExpArr", and the converted postfix expression is "PostfixExp", and operator stack is "mystack", the algorithm of converting infix expression to postfix expression can be described as follows:

Step 1: Initializing the mystack, and pre-process the InfixExp to InfixExpArr, and reading the first element of InfixExpArr;

Step 2: If read data is operand, appending it to the PostfixExp (when finishing the appending operation, also need to add a blank in the end of PostfixExp), and going to step 6;

Step 3: If read data is left parenthesis, pushing it to mystack and going to step 6;

Step 4: If read data is right parenthesis, appending element in mystack with a end of blank to PostfixExp in order until meeting first left parenthesis, and going to step 6; if left parenthesis can not be met after all elements in mystack have been pop, it means that left parenthesis does not match right parenthesis, and algorithm is end;

Step 5: if read data is operator, supposing the operator as "op", and checking the element of mystack in order, while the priority of an element in mystack is greater or equal than op, then popping the
element in mystack and appending it with a blank to PostfixExp, until meeting an operator in mystack whose priority is less than op, and finally going to step 6;

Step 6: if element is not read in the InfixExpArr, reading next element, and going to step 2; else algorithm is end.

After converting infix expression to postfix expression, value of postfix expression can be evaluated easily. We can partition postfix expression to operand and operator according to blank and store operand and operator with a string array: PostfixExpArr, and we suppose stack of operator is mystack1, and then the algorithm of evaluating postfix expression can be described as follows:

Step 1: Initializing the mystack1, and read first element from the PostfixExpArr;
Step 2: if read data is operand, pushing it into mystack1 and going to step 4;
Step 3: if read data is operator, supposing it as "op", and popping data from mystack1 twice, and suppose the two data as data1 and data2, and evaluate the Boolean expression: "data1 op data2", at last pushing the result into mystack1;
Step 4: if there is element is not read in the PostfixExpArr, reading a next element, and going to step 2; or else returning the top data of mystack1 as the result of the postfix expression and algorithm is end.

3. Design and implementation of improved Boolean expression evaluation

Above algorithm of Boolean expression evaluation has a severe problem that algorithm is not independent and can not expanded. The reason exists as follows:

The name, priority and operation rule of operator is embedded in the evaluation algorithm as the form of hard-coding, which results in that new operator can not be added and the operation rule of original operator can not be updated.

The algorithm of evaluation is not independent, and couples tightly with operator. To solve the above problem, the following method is adopted:

Encapsulating name, priority and operation rule of operator as the form of independent interface. User can define new operator class according to the standard interface and define name, priority and operation rule in this defined class. Otherwise, eight operator classes have been pre-defined: AndOperator, EqualOperator, GEqualOperator, GreatOperator, LEqualOperator, LowOperator, NotEqualOperator and OrOperator. The AndOperator class denotes operation of And, and the EqualOperator class denotes operation of Equal, and GEqualOperator denotes operation of Great Equal, the GreatOperator class denotes operation of Great, the LEqualOperator class denotes operation of Low Equal, the LowOperator class denotes operation of Low, the NotEqualOperator class denotes operation of Equal Not Equal, and the OrOperator class denotes operation of Or. By this method, the problem that name, priority and operation rule of operator is hard-coding in the algorithm is resolved.

Using operator object list to manage all operator object, and allowing user to insert a new operator object or update an original object, and allow user to query a special operator object by a given operator name. Above eight operator classes have create individually object and have been built in the operator object list. By this method, the problem that operator can not be new and updated is resolved.

Improving the algorithm of converting the infix expression to the postfix expression and the algorithm of postfix expression evaluation. In the improved algorithm of converting the infix expression to the postfix expression, getting the priority of operator must firstly query the operator object from the operator object list and then get the priority from the operator object, and judging a string as an operator according to query operator object list. Otherwise, considering the complexity of the operand of user-defined operator class, the lexical rules of literal is improved as follows: `<Literal>::= {(Literal (, Literal)*) (, Literal (, Literal)*)*} ` By this improvement, the data, which is placed within the bracket, is user-defined literal in which many pair of data can be allowed to define by user. Each pair of data is placed within a bracket and partitioned by semicolon, and there are many data, which are partitioned by comma, in a pair of data. In the improved algorithm of evaluating the postfix expression, getting the operation rule of operator must also query the operator object list and then get the operation rule from the operator object. By two improved algorithm, Boolean expression evaluation can not only be independent, but also is loose coupled with operator class.

The UML class diagram of the improve algorithm is shown as Fig. 1.

IBinaryOperator is the operator interface, and user can defined his own class to implement this interface. The abstract method defined in the interface is described as follows:
String getName(): getting the name of operator.
String getDescription(): getting the description of operator.
byte getPriority(): getting the priority of operator.
boolean match(Object oprnd1, Object oprnd2): judging two arguments whether is suited for the operator.
Object eval(Object oprnd1, Object oprnd2): evaluating the two arguments by operation rule of the operator.

Figure 1. UML class diagram of improved Boolean expression evaluation

IExpressionEvaluation is the core interface of the improved Boolean expression evaluation, and the abstract method defined in the interface is described as follows:
HandlePrefixExpr(String infixExpr): the method of pre-processing the infix expression and the prefixExpr argument is an infix expression.
addOpr(BinaryOperator opr): the method of adding operator object.
toPostFixExpr(String infixExpr): the method of converting infix expression to postfix expression.
eval(String postFixExpr): the method of evaluating postfix expression.
eval(Object opr1, Object opr2, String oprt): the method of evaluating the expression of "opr1 oprt opr2".

The ExpressionEvaluation class implements the IExpressionEvaluation interface. In the class, there are extra member are added and they are described as follows:

- oprList: Operator object list which is member variable.
- getOprPriority(String operator): the method of getting priority of the given operator.
- isOperand(String data): the method of judging data whether an operand.
- isOperator(String data): the method of judging data whether an operator.

The IBinaryOperator interface is the middle layer between operator class (for example, AndOperator class EqualOperator class, etc.) and IExpressionEvaluation interface, and decouples between them, and not only ensures the independence of expression evaluation algorithm, but also increases the scalability of the class of operators in order to achieve the scalability of the expression evaluation.

4. Special application and experiment

Based on above Boolean expression evaluation system, we can apply it to the information distribution system. In the information distribution system, the principle of information distribution within a region of the air intelligence is to judge whether the spatial coordinates of the air intelligence information received is in a special region. If the air intelligence information is in the region, the received two-dimensional coordinates of the air intelligence is a single point coordinates, such as: {(0.5, 0.5)}, which is denoted by oprnd1. Assuming that the operator name, determines whether the air intelligence is in a region, is "Polygon", you need to calculate the value of the Boolean expression: oprnd1 Polygon oprnd2. Operator class "PolygonOperator" is defined to realize the function, and the main algorithm is shown as Figure 2.

```
public class PolygonOperator implements IBinaryOperator {
    public Object eval(Object oprnd1, Object oprnd2) {
        //Calling the catch (oprnd1, oprnd2) to determine whether the parameters are valid
        if ( !(oprnd1 instanceof Double) || !(oprnd2 instanceof Double) ) {
            throw new NumberFormatException("Wrong parameter type!");
        } //Removing the header of oprnd1 and oprnd2
        String newOprnd1 = oprnd1.toString().replace("\", " ").replace("\", " ");
        String newOprnd2 = oprnd2.toString().replace("\", " ").replace("\", " ");
        //Using Java.net.geom.Origin class to create space region object: polygon
        GeneralPath polygon = new GeneralPath();
        //Splitting the newOprnd1 into a string array
        String[] polygonArr = newOprnd1.replaceAll("\", " ").replaceAll("\", " ");
        //Putting all coordinates into polygonPath in order to set the polygon region
        polygon.moveTo(new Double(polygonArr[0]), new Double(polygonArr[1]));
        for (int i = 2; i < polygonArr.length; i += 2) {
            polygon.lineTo(new Double(polygonArr[i - 1]), new Double(polygonArr[i]));
        } //Closing the special space region
        polygon.closePath();
        //Splitting the newOprnd2 into a string array
        String[] polygonArr2 = newOprnd2.replaceAll("\", " ").replaceAll("\", " ");
        //Calculating the area of two polygonPath
        polygonArea = polygonArea + new PolygonPath(newOprnd1, newOprnd2, polygonArr, polygonArr2);
    }
    public String getDescription() {
        return "Calculating whether points is in the set area";
    }
    public String getName() { return "Polygon"; }
    public byte getPriority() { return 21; }
    public boolean catchObject(Object oprnd1, Object oprnd2) {
        String newOprnd1 = oprnd1.toString();
        String newOprnd2 = oprnd2.toString();
        if (newOprnd1.length() > 2 && newOprnd2.length() > 2)
            return true;
    }

    //Calculating the area of two polygonPath
    polygonArea = polygonArea + new PolygonPath(newOprnd1, newOprnd2, polygonArr, polygonArr2);
}
```

Figure 2. Algorithm of Polygon Operator
In addition, the information distribution system needs to modify the operation rule of greater than operator as follows: assume that A is greater than B, then the mathematical description becomes $A > B + 1$. So we need to define a new operator class to inherit the original GreatOperator class and its main changes of algorithm are shown as Figure 3.

```java
public class NewGreatOperator extends GreatOperator {
    public Object eval(Object oprnd1, Object oprnd2) {
        // Calling the match (oprnd1, oprnd2) to determine whether the parameters are valid
        if (!match(oprnd1, oprnd2)) {
            throw new NumberFormatException("Data format error");
        }
        return Double.parseDouble(oprnd1.toString()) + Double.parseDouble(oprnd2.toString()) + 1;
    }
}
```

Figure 3. Algorithm of a New Great Operator

In order to validate the function of the new operator "Polygon", test points and Region, which are shown as Table 1, are provided.

<table>
<thead>
<tr>
<th>Test Point</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0.5, 0.5}</td>
<td>{(0,0), (0,1), (1,1), (1,0)}</td>
</tr>
<tr>
<td>{0.9, 0.9}</td>
<td>{(0,0), (0,1), (1,1), (1,0)}</td>
</tr>
<tr>
<td>{1.5, 0.5}</td>
<td>{(0,0), (0,1), (1,1), (1,0)}</td>
</tr>
<tr>
<td>{0.5, 1.5}</td>
<td>{(0,0), (0,1), (1,1), (1,0)}</td>
</tr>
<tr>
<td>{1.5, 1.5}</td>
<td>{(1,1), (2,1), (2,2), (1,2)}</td>
</tr>
<tr>
<td>{1.9, 1.9}</td>
<td>{(1,1), (2,1), (2,2), (1,2)}</td>
</tr>
<tr>
<td>{2.5, 0.5}</td>
<td>{(1,1), (2,1), (2,2), (1,2)}</td>
</tr>
<tr>
<td>{0.5, 2.5}</td>
<td>{(1,1), (1,2), (2,2), (1,2)}</td>
</tr>
</tbody>
</table>

Table 1. Test Data of polygon operator

Based on the above test data, test code is shown as Figure 4 and run result is shown as Figure 5.

```java
public class NewPolygonOperator {
    public Object eval(Object expr) {
        ExpressionEvaluator exprEval = new ExpressionEvaluator();
        GreatOperator op = new PolygonOperator();
        String[] region = {"[0.5, 0.5], [0.9, 0.9], [1.5, 0.5], [0.5, 1.5], [1.5, 1.5], [1.9, 1.9], [2.5, 0.5], [0.5, 2.5]"};
        for (int i = 0; i < region.length; i++) {
            if (Boolean.valueOf(exprEval.evaluate("region", 0).tostring())) {
                System.out.println("is in " + region[i]);
            } else System.out.println("is not in " + region[i]);
        }
    }
}
```

Figure 4. Test code of Polygon Operator
In order to validate the function of the new function of ">\"", test data, which are shown as table 2, are provided.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>16</td>
</tr>
</tbody>
</table>

Based on the above test data, test code is shown as Fig. 6 and run result is shown as Figure 7.

```java
public class Main {
    public static void main(String[] args) {
        ExpressionEvaluator expEval = new ExpressionEvaluator();  
        BinaryOperator po = new NewBinaryOperator();  
        expEval.addOper(po);  
        String[] data1 = {"12","13","14","15","16","17","18","19"};  
        String[] data2 = {"13","13","13","13","16","16","16","16"};  
        for(int i = 0; i < data1.length; i++) {
            if (Boolean.valueOf(expEval.eval(data1[i] + " > " + data2[0].toString()))) {
                System.out.println(data1[i] + " is not great " + data2[0]);
            } else {
                System.out.println(data1[i] + " is great " + data2[0]);
            }
        }
    }
}
```

**Figure 6.** Test code of Great Operator

**Figure 7.** Running result of Great Operator
5. Conclusions

In this paper, according to the research of the traditional Boolean expression evaluation algorithm, the following problems have been found out: the algorithm is not independent because the name, priority and operation rule of operator are hard-coded in the algorithm, which causes the algorithm is difficult to extend. Aim at above problems, a new kind of Boolean expression evaluation system has been advanced, in which the operator (including its name, priority, and operation rule) has been subtracted as the form of interface from the original Boolean expression evaluation algorithm, and has decoupled them using a middle layer. The experiments have shown that the algorithm is an extensible Boolean expression evaluation algorithm. Also, there is a lot of room for improvement, including further optimizing the algorithm in future.

6. Acknowledgment

This work was supported by the technology of high dynamic X-ray Digital imaging based on self-adaptive dose (No.61071193).

7. References