Paper 25-26

Storing and Resolving Dynamic Formulas

Chris D. St.Peter, Technology Professionals Corporation, Grand Rapids, MI

ABSTRACT

Often programmers need to perform custom calculations to display the results in their output. The most common way to do this is probably by embedding hard-coded formulas into the code. This is fine if there is no chance that these formulas will change. However, what happens when formulas get really complex? What if they need to change over time, or can be nested? This paper explores a way to store formula information and resolve those formulas using Version 8e (SAS/AF, SCL and base SAS). This paper is intended for advanced programmers with a basic knowledge of object-oriented programming.

INTRODUCTION

Many times our SAS® work requires us to use formulas to calculate numbers we need for a report or some other output. Sometimes they need to be dynamic. They may also need to contain other dynamic formulas. This paper explores some ways that can be used to store information about dynamic formulas and resolve them using SAS/AF® V8e and SCL. These points are illustrated using a simple example. Then, the paper suggests a few ways that these ideas can be expanded to accommodate more complicated situations.

STORING FORMULAS

Since we are mostly concerned with the storing and resolving of formulas, this paper will not go into detail on the appropriate interface to collect formula information. However, as an example, you can use SAS/AF to create an interface through which the user can enter formulas. Whatever the interface, it should control which sorts of operators and operands can occur successively (for example, follow every operand but the last one with an explicit mathematical operator). It should only allow committing when the formula is at a steady state (matching parentheses, not ending with an operator that is not a parenthesis, etc.). Further, the interface should prevent a formula from containing itself or any formula which contains it to prevent an endless loop at resolution time.

To effectively store the formula information, we will break it down into the smallest logical units. We will refer to these as tokens. Tokens can come in several forms, including the following:

- *Operators*:
  Parentheses and mathematical operators will belong to the operator type.

- *Numbers*:
  Any numeric values will be considered as the number type.

- *Variables*:
  We will include variables in our example as a type of token. For the sake of simplicity in the example, we will assume that all formulas relate to information stored in a single SAS data set. Further, we assume that the interface allows you to choose any of the numeric variables in the data set to include in your formula. We also assume that any variable chosen will resolve to the sum of that variable over the entire data set. (These assumptions can be altered to fit your real-world application, but are used here for conceptual ease).

- *Formulas*:
  Formulas can contain other formulas, so a nested formula is another type of token.

- *Custom:*
  You can add custom types that represent certain variables from other data sets or anything that you can define a custom accessor method to retrieve. This type can be used when data retrieval is not

straightforward (e.g. it requires a join or represents a ratio of two variables).

For the sake of our example, we will include the first four token types: Operators, Numbers, Variables and Formulas.

We choose to normalize the data to keep the tokens separate from descriptive information about the equation as a whole. This approach leads us to use two data sets to store the formula information (the names of the key variables are in italics). One holds the overall equation information:

EQUATION data set

Variable	Type	Length	Label
Eq_ID	Number	8	Equation ID
Eq_Desc	Text	40	Equation Description

And another one stores the specific token information:

TOKENS data set

Variable	Type	Length	Label
Eq_ID	Number	8	Equation ID
Seq_No	Number	8	Sequence Number
Token_Text	Text	256	Token Text
Token_Type	Text	1	Token Type
Token_Eq_ID	Number	8	Token Equation ID

Before we examine the code we will use to manipulate these data sets, let's examine a sample case and what its data would look like. Assume we have three formulas as follows:

Constant = 12

Formula1 = 34 * *Constant*

Formula2 = *hours_worked* * (*Formula1* / 8)

Where *hours_worked* represents the summing of the variable by that name in a table called *work.actual_hours*.

These three formulas would be represented in the *EQUATION* table as follows:

Eq_ID	Eq_Desc
1	Constant
2	Formula1
3	Formula2

They would be represented in the *TOKENS* table as follows:

Eq_ ID	Token_ Text	Token_ Type	Token_ Eq_ID	Seq_ No
1	12	N	.	0
2	34	N	.	0
2	*	O	.	1
2		F	1	2
3	hours_ worked	V	.	0
3	*	O	.	1
3	(	O	.	2
3		F	2	3
3	/	O	.	4
3	8	N	.	5
3	)	O	.	6

As you can see, for every type of token, the *eq_id* variable contains the ID of the equation that the token belongs to. Also, the *seq_no* variable always holds the token's sequence in the equation. Now we need to examine how the other variables in this data set are used for our four types.

- *Operators:* The *token_text* will be the operator itself (for example '+'), the *token_type* will be 'O' for operator and the *token_eq_id* will be null.
- *Numbers:* The *token_text* is the number and the *token_type* is 'N'. Again, the *token_eq_id* is null.
- *Formulas:* The *token_text* is null (the formula name is held in the *EQUATION* table). The *token_type* is 'F' and the *token_eq_id* is the equation ID of the contained formula.
- *Variables:* The *token_text* is the name of the variable. The *token_type* is 'V' and the *token_eq_id* will be null.

RESOLVING FORMULAS

Now that we have looked at how the formulas are stored, we need to focus on how that information is used to resolve the formulas. Since formulas can be nested, the process lends itself to a recursive solution. The following SCL method can be used to resolve formulas. (Following the sample code, there is an explanation for each numbered line.)

```
resolveFormula: /*resolveFormula method*/
Public Method
1  equationId_n   : input : num
2  eq_dsid        : input : num
3  return = num
;
   declare char (1)
      tokenType_c
   ;

   declare char (100)
      tokenText_c
      valueVar_c
      varName_c
      whereClause_c
   ;

   declare char (1000)
      formulaText_c
   ;

   declare num
      ignored_n
      newValue_n
      outValue_n
      tokenEquationId_n
   ;

   outValue_n = .;

   /* Subset the input data set based on
      the input equation ID */
4  whereClause_c = 'EQ_ID=' ||
      compress(put(equationId_n, best.));

   /* If there is already a where clause
      in effect, add the word also to the
      front of this clause so it doesn't
      override any existing clause */
   if attrn(eq_dsid, 'whstmt') >= 2
         then do;
      whereClause_c = 'also ' ||
         whereClause_c;
   end;
   ignored_n = where(eq_dsid,
      whereClause_c);

   /* Cycle through the tokens for this
      equation */
   i = 1;
5  do while (fetchobs(eq_dsid, i) = 0);
      i + 1;
      tokenType_c = getvarc(eq_dsid,
         varnum(eq_dsid, 'token_type'));

      /* Resolve each token into a number
         or operator */
      select tokenType_c;
6        when ('F') do;
            tokenEquationId_n =
               getvarn(eq_dsid,
               varnum(eq_dsid,
               'token_eq_id'));
            /* Undo the current where
               clause */
7           ignored_n =
               where(eq_dsid, 'undo');
            /* Call the resolveFormula
               method recursively */
8           newValue_n =
               _self_.resolveFormula
               (tokenEquationId_n,
               eq_dsid);
            tokenText_c =
               compress(put(newValue_n,
               best.));
            /* Replace where clause */
9           ignored_n =
               where(eq_dsid,
               whereClause_c);
         end;
10       when ('V') do;
            /* Call method to retrieve
               sum of variable from
               data set */
            varName_c =
               getvarc(eq_dsid,
               varnum(eq_dsid,
               'token_text'));
            tokenText_c =
               _self_.getVarSum
               (varName_c);
         end;
11       otherwise do;
            tokenText_c =
               getvarc(eq_dsid,
               varnum(eq_dsid,
               'token_text'));
         end;
      end;

12    formulaText_c = formulaText_c ||
         tokenText_c;
   end;

   /* Undo the current where clause */
13 ignored_n = where (eq_dsid, 'undo');

   /* Create a global macro variable to
      receive the value */
14 valueVar_c = 'valu_var';
   submit continue;
      %let &valueVar_c =
         %sysevalf(&formulaText_c);
   endsubmit;

15 outValue_n = input(compress
      (symget(valueVar_c)), best.);

   return outValue_n;
endmethod;
```

1. The first input parameter is the ID of the equation to resolve.

2. The second input parameter is the data set ID of the equation data set (already opened in a read-only format). We pass in the data set ID of the equation data set instead of just opening it in the method for the sake of flexibility. Sometimes (particularly when we are

hitting a data source other than SAS) we may have issues where each hit to the data source costs a lot of time, so we want to retrieve the information for all equations at once. In this case, the caller can subset that data set before sending in its ID so that we only have information for the overall equation we are resolving (in this case Formula2) and all its contained formulas (in our example, Formula1 and Constant). The method may end up applying and removing where clauses many times (due to its recursion). In doing so, it never eliminates where clauses that may have been put on before the method was called. This allows the method to be usable whether it is looking at an entire data set, or one that was previously subset.

3. The returned value is the resolved value of the formula (a single number).

4. We subset the data set by the input equation ID. We first need to check for existing temporary where clauses (for more information on the return codes from the ATTRN statement with the 'whstmt' attribute, see *SAS Screen Control Language: Reference, Version 6, Second Edition*, p.239). If we find that a temporary where clause exists, we need to add "also " to the front of our where clause so we don't lose the existing where clause.

5. After, that, the method cycles through the observations for this equation. It needs to use *fetchobs* because the recursive call requires undoing and then resetting the where clause. Adjusting the where clause will reset the pointer in the data set, so the method needs to be able to fetch the observation it left off on rather than starting again at the beginning of the data set.

6. Resolve nested formulas to a single number. Start by retrieving the ID of the nested equation.

7. Undo the last where clause that we applied for this call to the method.

8. Call the *resolveFormula* method recursively. To do this and also take advantage of Version 8's compile time binding and dot notation, we need an extra step in a part of the SCL code that is not shown. By default, the *_self_* variable is declared for you, but it is declared as the generic 'object' type. That means you can use dot notation with it, but if you want to take full advantage of compile time binding with the *_self_* object, you need to declare it as an instance of the class where the method resides. For example, if our *resolveFormula* method is defined on a class called *Form_Res*, we would include this statement in our code:

```
declare      Form_Res      _self_;
```

   If we include this line in our example, that means that, among other things, SAS would check at compile time to be sure our recursive call has the right signature.

9. After the method has the value returned from the recursive call, it needs to reapply the where clause. This is the point at which it becomes important to use *fetchobs* as opposed to *fetch*. Reapplying the where clause has moved the pointer back up to the first observation in the subset, but the method must pick up where it left off before the recursive call.

10. For variables, we retrieve the variable name from the *token_text* variable in our *TOKENS* data set. Then, we call a method (*getVarSum*) which returns the sum of that variable from the data set that this formula relates to. As mentioned earlier, the data set is not important, nor are the details of how the sum is retrieved. For our example it is enough to know that the *getVarSum* method returns the sum of the named variable.

11. For tokens other than variables and formulas, we simply add their text onto the *formulaText_c* variable. This action is based on the assumption that the user interface has properly restricted the order and type of these tokens.

12. We need to make sure that the length of the *formulaText_c* variable is enough to accommodate all the tokens that will be appended to it. If this were implemented in 6.12, we would need to be aware of the 200 character limit and find a slight adaptation of this methodology to be sure we didn't lose information because our variable was not big enough to handle it. One way to do that would be to put each token into a list (resolving all nested formulas into a single number as above) and regurgitate that list into a *%sysevalf*, in a way similar to how this method builds the *formulaText_c* variable.

13. Undo the where clause we implemented at the beginning of the method.

14. Set up a macro variable to receive the results of performing the *%sysevalf* function on the *formulaText_c* variable.

15. Retrieve the value of that macro variable and then return it as the value of the function.

POSSIBLE ENHANCEMENTS

There are many possibilities to expand on this methodology. Since it is often required in real-world reporting applications, I would like to focus on enhancements related to formulas that calculate results for a given time period. Once you start dealing with time periods, a lot of possibilities open up:

- You can significantly expand the way your formulas use variables from SAS data sets or from tables in other databases.
- You can maintain versions of the formulas which have effective dates.
- You can allow rolling averages and other similar measures.

Using variables more extensively

Some possible enhancements in this area include: allowing the user to choose the table as well as the variable for more flexibility. Also, if you are reporting over a time period, you can use a date variable in the data set so that you only grab data that was logged within your reporting period. Depending on how complex the way of getting or interpreting variables is, you can have one method to retrieve variables (as in the example) or several. That is also a good place to use object-oriented programming. For example, you could have a data retrieval class and then define a subclass for each data source you need to get information from. That way you can call the same method no matter which object you have an instance of. That object will have its method defined to get the necessary data from its particular data source. These kinds of enhancements really start to show the power of this methodology because you now have access to all kinds of different data, subsetted over time, with nearly limitless possibilities.

Versioning with effective dates

So far, we have been assuming that when formulas change, we can overwrite the old one and always use the new one, even if we are running the formula on a previous time period. Sometimes it is necessary to maintain the older versions as the formula changes. One of the main reasons for this is encountered when the formula contributes to a report. It may be necessary to run legacy reports that match the ones run with the previous version of the formula. If that is the case, we need to have a record of all previous versions of the formula. To accomplish this, we would assign effective dates to each equation. When that equation changes, rather than overwriting it, we maintain the previous version(s) and create a new one with a new effective date. If a reporting period spans multiple effective dates, the data would need to be separated out by effective date ranges. Then, an algorithm would have to be determined for how to deal with this data (e.g. is it a straight sum, is it a rate that needs to be weighted by time period or number of observations?)

Rolling Averages and other measures

This concept is based on allowing data to be used which goes beyond the restrictions of a given reporting time period. Instead of just gathering data for whatever the report period is, you could alternatively specify a certain amount of time in the past from the report's ending date. For example, you can allow a monthly report to contain a 12 month rolling average. In this case, the data for most formulas in the report would be restricted to the time period of the report, but this measure would span the last twelve month's worth of data. This allows comparisons of measures from different time periods.

CONCLUSION

When formulas become complex and dynamic, it is necessary to accommodate them with a structured approach. The approach described here has many advantages. It centralizes your formula information. It allows you to change the formulas without changing code. It addresses issues such as nesting and is expandable to deal with reporting time periods and other complex issues. This solution can be adapted to fit many situations and it applies particularly well to formulas used in reports.

REFERENCES

McConnell, Steve (1993), *Code Complete: A Practical Handbook of Software Construction*, Redmond, WA: Microsoft Press

SAS Institute Inc (1995*), SAS Language: Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc (1994*), SAS Screen Control Language: Reference, Version 6, Second Edition*, Cary, NC: SAS Institute Inc.

ACKNOWLEDGEMENTS

I would like to acknowledge the contributions of Jack Fuller (also of TPC) to the concepts covered in this paper.

The author may be contacted at:

Chris D. St.Peter
Application Developer
Technology Professionals Corporation
1 Ionia SW, Suite 400
Grand Rapids, MI 49503
cdstpete@teamtpc.com