Data Without (Step) Boundaries: Using Data Access Functions
Felix Galbis-Reig, George Washington University Biostatistics Center, Rockville, MD

ABSTRACT
This paper explores the use of data access functions to retrieve and manipulate data stored in a SAS data set, with the main goal being to demonstrate possible uses of this alternative access scheme. To this end I will motivate the use of these methods by providing some simple examples. The reader should be familiar with basic DATA step concepts and have some experience with the SAS Macro facility.

INTRODUCTION
Within the context of a DATA step and its implied looping structure, SET, MERGE, INPUT, and related data access statements provide a straightforward and flexible means of performing almost any data access task that you are likely to come across. In certain situations, however, it is convenient to access the data in a data set without using this familiar construct. In some cases, accessing data sets by the traditional route might result in complicated or convoluted code structures or may lead to more data manipulation than is necessary. When writing general purpose Macro code, it may prove useful to access the data contained in a data set without creating or forcing any step boundaries.

I will provide two examples modeled on problems encountered in my own work where foregoing the use of data access statements and controlling data access directly using data access functions has proved useful. In the first example we are given a data set containing observations that include variables whose values reference variables in another data set. The task is to extract, for each observation in the first data set, the values of the referenced variables from the second data set. In the second example, we define a macro that can be called within a DATA step and uses information contained in a data set to programmatically create data-dependent conditional logic for the parent DATA step.

A FIRST LOOK: MIMICKING A SET STATEMENT
If you have no experience using data access functions, the best way to begin to acquire an understanding of their use is to see them in action performing a familiar task. In what follows, I use data access functions to reproduce the behavior of the SET statement used in a DATA step. Consider the following code:

```
DATA OUTPUTDS;
  SET INPUTDS;
RUN;
```

This simple DATA step uses the SET statement to read a data set named INPUTDS and creates a second data set named OUTPUTDS that is equivalent to INPUTDS in terms of both structure and contents. This same result can be achieved using data access functions as follows:

```
DATA OUTPUTDS (DROP=DSID RC);
 1   IF (0) THEN SET INPUTDS;     /* unexecuted SET statement */
 2   DSID=OPEN("INPUTDS");       /* OPEN the data set */
 3   CALL SET(DSID);              /* use CALL SET to get variable values */
 4   RC=FETCH(DSID);              /* FETCH the first observation */
 5   DO WHILE (RC=0);             /* while the last FETCH retrieved an obs */
       OUTPUT;                    /* OUTPUT an observation */
       RC=FETCH(DSID);            /* FETCH the next observation */
   END;
 6   RC=CLOSE(DSID);              /* CLOSE the INPUTDS data set */
RUN;
```

This code may look a little daunting at first but as you will soon see it is actually straightforward. Before attempting to explain this code in detail, I will first lay a foundation by presenting some basic concepts associated with data access in the SAS framework.
BASIC CONCEPTS

In the SAS environment it is helpful to think of data access as consisting of four tasks: accessing or “opening” the data source (which for the purposes of this paper will be a SAS data set), retrieving or “fetching” a particular observation, “getting” or “setting” the values into program variables (DATA step or macro variables), and releasing or “closing” the data source. This process is summarized in the figure below.

The following sections briefly describe these tasks and the data access functions the SAS language provides to accomplish each. For a more detailed explanation and some usage notes for the functions referenced in this paper, please refer to the Function Reference Table at the end of the paper.

“OPENING” A SAS DATA SET

In SAS we use the OPEN function to access or “open” an existing data set. At a minimum, a call to the OPEN function should specify the name of the data set to be opened. At the time the data set is opened a data set data vector (DDV) is created. Similar conceptually to the program data vector (PDV) in DATA steps, the DDV is a logical area in memory that is used to hold an observation from the opened data set. The OPEN function returns a unique numeric data set identifier that we can use in subsequent function calls as a handle or reference to the opened data set. Once you have obtained a valid data set identifier you can use it to fetch observations from the data set and load them into the DDV.

“FETCHING” AN OBSERVATION

Once a data set has been opened we use either the FETCH or FETCHOBS function to read an observation from the data set into the DDV. A call to FETCH loads the next observation in the data set and requires a single argument; a data set identifier that was returned by a call to OPEN. In contrast the FETCHOBS function allows you to specify the observation number that should be loaded into the DDV and so a call to FETCHOBS requires two arguments; a data set identifier and a value that specifies the observation number that should be loaded.

“GETTING”/“SETTING” VALUES

After fetching a data set observation and loading it into the DDV, the next step is to transfer data values from the DDV to the program variables (data step or macro variables) where they will be used. The SAS language provides two ways to perform this task.

Using the GETVARN or GETVARC functions we can manually extract the values of individual variables and pass them directly to program variables using a simple assignment. Each of these functions takes two arguments: a data set identifier and a number that identifies the variable of interest. The variable is identified by its position in an observation. The VARNUM function provides an easy way to get the variable position given a variable name. As the names imply, the only difference between these two functions is that the GETVARN is used to get the value of a numeric variable whereas the GETVARC is used to return the value of a character variable.

You can also use the SET call routine to automatically transfer values from the DDV to corresponding program variables immediately after an observation is loaded into the DDV. In order for variables to be automatically transferred, it is important to define program variables that match the data set variables with respect to name (and type if dealing with DATA step variables). If a corresponding program variable has not been defined for a data set variable in the DDV, the routine does not create a program variable for you.

“CLOSING” A SAS DATA SET

Once you have finished using a data set, it is important to “close” the data set using the CLOSE function to release any resources or locks associated with its use. If the data set was opened within a data step, it will be automatically closed when the data step terminates so that it is not necessary to call the CLOSE function explicitly. It is still a good idea to get into the habit of doing this yourself since this behavior is not automatic in other contexts.
A SECOND LOOK

We are now in a position to examine the code introduced earlier that reproduces the behavior of a SET statement. This relatively simple block of code performs all of the basic tasks that were described in the previous section. In the discussion that follows, the numbers refer to the numbering found to the left of the code. The data set (INPUTDS) is “opened” (2) and the first observation is “fetched” (4). Because I have used the SET call routine (3), when an observation is fetched the values loaded into the DDV are automatically transferred to corresponding program variables. In this case, I have used an unexecuted SET statement (1) as a convenient way of ensuring that every variable defined in INPUTDS is defined as program variable. (For an explanation of the use of an unexecuted SET statement refer to the paper by Fuller referenced at the end of the paper.) While the last FETCH operation was successful, I write the contents of the PDV to the output data set using the OUTPUT statement and fetch the next observation. We continue in this way until there are no observations left to read from the input data set. As a final step I make sure to close the open data set (6).

At this point you might be thinking to yourself that, given that these two approaches accomplish the same end, why would you ever choose to use the second approach? In certain situations, the alternative approach gives you added flexibility that you can leverage to accomplish a task that would otherwise prove to be difficult. Notice that we now have an additional copy of the current observation stored in memory (in the DDV). This additional copy gives us an alternative means to access the data. In particular, as you will see in the following example, we can use the VARNUM function to access a particular variable using a name that is dynamically provided as data.

EXAMPLE 1 (USING ACCESS FUNCTIONS IN A DATA STEP)

Consider two datasets with structures as indicated in the figure below.

<table>
<thead>
<tr>
<th>SOURCEDS</th>
<th>LOGDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>VAR1</td>
<td>V_NAME</td>
</tr>
<tr>
<td>VAR2</td>
<td>VALUE</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>VAR1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>VAR2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>VAR2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In SOURCEDS, the variable ID uniquely identifies an observation and the remaining variables contain data pertaining to that observation. In LOGDS, ID is the unique identifier of an observation in SOURCEDS, V_NAME holds the name of a variable from SOURCEDS, and VALUE is a character representation of the value of the variable referenced by V_NAME at a given point in time. This particular structure can arise naturally in the context of a data validation system in which we log any data issues that have been identified. Given this set up, if we are operating in a transactional environment where the data stored in SOURCEDS is subject to possible update it might be useful to compare the (historical) value stored in the logging data set with the current value stored in the source data set. What we would ideally like to have is a data set similar in structure to LOGDS that is augmented by the current value. This would allow us to easily check for cases in which the data has changed since an issue was identified particularly in cases where the value referenced in LOGDS was verified as correct.

The apparently simple task described above is more difficult than it would at first appear to be. As a first thought it might seem that we would achieve our goal by simply merging the source data set back into the logging data set. Even after performing this merge, however, we would not be done; we would still need to find a way to extract the value of the variable referenced by V_NAME. Even though it seems like this should be easy or at least possible in the data step used to perform the merge (after all we have all the information we need in a single observation in the PDV), it turns out that there is no way to do this directly. Although we can conceivably perform this task by restructuring SOURCEDS before merging it back into LOGDS by ID and V_NAME, this solution does not generalize well to the case where LOGDS contains more than one variable reference per observation. It turns out that with only a slight modification the code used to mimic a SET introduced above provides an elegant and efficient solution to this problem. The key is that we can return the value of a variable whose name is provided by the value of another variable.

3
DATA OUTPUTDS (KEEP=ID V_NAME VALUE VALUE_CUR);
   IF (0) SET INPUTDS;
   LENGTH VALUE_CUR $ 15;
   DSID=OPEN("INPUTDS");
   CALL SET(DSID);
   RC=FETCH(DSID);
   DO WHILE (RC=0):
1     VNUM=VARNUM(DSID,V_NAME); /*get position for var referenced by V_NAME */
2     VTYPE=VARTYPE(DSID,VNUM); /*get variable type */
3     VFORMAT=VARFMT(DSID,VNUM); /*get variable format */
4     IF VTYPE='N' THEN DO; /*if numeric call GETVARN and put value */
5       VALUE_CUR=PUTN(GETVARN(DSID,VNUM),VFORMAT);
6     ELSE VALUE_CUR=GETVARC(DSID,VNUM);
   OUTPUT;
   RC=FETCH(DSID);
   END;
   RC=CLOSE(DSID);
RUN;

The bulk of this code should look familiar to you from the last example. Note that in this case INPUTDS refers to the data set obtained by merging the logging data (LOGDS) set with the source data set (SOURCEDS) by ID. It follows that augmenting the logging data set with the current values is still a two-step process. Although it is possible to reduce this problem to a one step process, doing so would necessitate using data access functions to perform the merge manually. Although an interesting exercise, this is beyond the scope of the current presentation and probably "overkill" for most applications.

The lines of code that have been added retrieve the value of the variable referenced by V_NAME from the DDV and transfer it (converted to a character string) to the new program variable VALUE_CUR. (Note that the value is converted to a character string so that VALUE_CUR will be able to hold the value associated with any data type.) Using the VARNUM function (1) we get the variable number of the variable whose name is given by the value of the variable V_NAME. This number is subsequently used in calls to the VARTYPE (2) and VARFMT (3) functions to obtain the type and format of the variable referenced by V_NAME. We next check the variable type (4). If the variable type is numeric, we use nested function calls to retrieve the value of the referenced variable from the DDV using the GETVARN function and insert the value into the program variable VALUE_CUR using the PUTN function and the format obtained in (2). If the variable is nonnumeric we use the GETVARC function and directly assign its value to VALUE_CUR.

This example hopefully demonstrates that even within a DATA step, foregoing the usual data access statements such as SET and MERGE and controlling the data access directly can be advantageous. The use of these functions is not confined to DATA steps however, and in the next example we will explore their use in Macro code.

EXAMPLE 2 (USING ACCESS FUNCTIONS IN MACRO CODE)
Using %SYSFUNC we can extend the use of data access functions to Macro code. The main advantage in this context is that we can access the data and descriptor information stored in a SAS data set without forcing step boundaries. This allows us to define data-driven macros that can be used inside other steps. Consider a dataset with the following structure

<table>
<thead>
<tr>
<th>RANGEDS</th>
<th>VAR_NAME</th>
<th>HI_VALUE</th>
<th>LOW_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VAR1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>VAR2</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>VAR3</td>
<td>20</td>
<td>30</td>
</tr>
</tbody>
</table>
Such a dataset can be used as a lookup table for a simple range checking system. In particular, we might like to use a data set of this form to generate conditional logic that can be used within a DATA step to perform range validations. The DATA step in question might produce an output data set similar in structure to the LOGDS from the previous example. Importantly, the information in the RANGEDS can be dynamic; user specified and/or data dependant (perhaps generated as the output of a PROC FREQ or PROC UNIVARIATE). What we would like to be able to do is construct a macro, %DOCHECK, such that the following DATA Step produces an output data set matching the LOGDS in structure.

```
DATA ERRORDS;
  SET INPUTDS;
  %DOCHECK;
RUN;
```

Since the %DOCHECK macro will be called from within a DATA step, we must ensure that the macro does not contain or force any step boundaries. What follows is a simple implementation of the %DOCHECK macro:

```
%DOCHECK;
1 %LOCAL VAR_NAME HI_VALUE LOW_VALUE I DSID RC NOBS;/* define program var */
2 %LET DSID = %SYSFUNC(OPEN(RANGEDS));              /* open RANGEDS */
3 %SYSCALL SET(DSID);                               /* link prog vars     */
4 %LET NOBS=%SYSFUNC(ATTRN(&DSID,NOBS));            /* get NOBS attribute */
5 %FOR I=1 %TO &NOBS;                               /* for each obs       */
6       %LET RC = %SYSFUNC(FETCHOBS(&DSID,&I));         /* FETCHOBS           */
7      IF &VAR_NAME<&LOW_VALUE OR &VAR_NAME>&HI_VALUE THEN DO; /* outpt logic*/
V_NAME="&VAR_NAME";
VALUE=&VAR_NAME;
OUTPUT;
    END;
8 %LET RC=%SYSFUNC(CLOSE(&DSID));                   /* close RANGEDS      */
%MEND;
```

This macro definition follows the same basic pattern we have seen in the previous code examples. I begin by defining some local macro variables (1). Notice that for each variable in the RANGEDS I have defined a corresponding macro variable. This is done so that I can use the SET routine (3) to automatically transfer the values from the DDV to macro variables when an observation is fetched. Next RANGEDS is opened (2). Note that in macro code, we do not need to enclose the data set name in quotes. In (4) I use a call to the ATTRN function to access the data set descriptor information and retrieve the number of observations contained in the data set. This value will be used in the FOR loop (5) that I use to fetch observations from RANGEDS using FETCHOBS in (6). This macro could just as easily been written using FETCH and the control structure used in the previous examples but I thought it would be helpful to present an example of a typical use of FETCHOBS. After fetching an observation, we generate the text that will be written into the DATA step. This process is repeated for each observation of RANGEDS and the data set is closed (8) after all the observations have been processed.

When this Macro is used with the data provided in the table, it will add the following statements to the DATA step generating ERRORDS:

```
IF VAR1<1 OR VAR1>5 THEN DO;
  V_NAME="VAR1";
  VALUE=VAR1;
  OUTPUT;
END;
IF VAR2<10 OR VAR2>20 THEN DO;
  V_NAME="VAR2";
  VALUE=VAR2;
  OUTPUT;
END;
...```
Note that in this example I have not bothered to convert the variables to character strings before assigning them to the variable VALUE simply because I have assumed that all variables being checked are numeric. In needed you could obtain the variable types and formats associated with particular variables and use them to modify the generated statements (along the lines of what was done in the first example). In general, the macro code presented above represents a structure that can easily be extended to handle more complex situations.

CONCLUSIONS

I hope that after reading this paper you have developed an appreciation for the usefulness of data access functions in SAS code, whether in a DATA step or when writing data-driven macros. As I mentioned in the introduction, the examples presented in this paper are simplified representations of problems I encountered in my own work. I found the techniques discussed in this paper extremely useful in these cases and the concepts presented here should give you an alternative point of approach for problems you may face in your own work.

FUNCTION REFERENCE

Function and routines referenced in this paper; please refer to SAS documentation for further details.

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>USE TO…</th>
<th>SYNTAX</th>
<th>EXAMPLE USE (D: DATA Step, M: Macro)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>Open a SAS data set</td>
<td>OPEN(&lt;data-set-name,&lt;mode&gt;&gt;)</td>
<td>D: DSID=OPEN(&quot;INPUTDS&quot;)&lt;br&gt;M: %LET DSID=%SYSFUNC(OPEN(INPUTDS));</td>
</tr>
<tr>
<td>CALL SET</td>
<td>Link data set variables to corresponding program variables</td>
<td>CALL SET(data-set-id)</td>
<td>D: CALL SET(DSID);&lt;br&gt;M: %SYSCALL SET(DSID);</td>
</tr>
<tr>
<td>GETVARN</td>
<td>Retrieve the value of the specified variable from the DDV and assign it to a program variable</td>
<td>GETVARN(data-set-id, var-num)</td>
<td>D: VAR1=GETVARN(DSID,1)&lt;br&gt;M: %LET VAR1=%SYSFUNC(GETVARN(DSID,1));</td>
</tr>
<tr>
<td>GETVARNC</td>
<td>Retrieve the value of the specified variable from the DDV and assign it to a program variable</td>
<td>GETVARNC(data-set-id, var-num)</td>
<td>D: VAR1=GETVARNC(DSID,1)&lt;br&gt;M: %LET VAR1=%SYSFUNC(GETVARNC(DSID,1));</td>
</tr>
<tr>
<td>VARTYPE</td>
<td>Return the corresponding variable attribute given the variable’s position in a data set.</td>
<td>VARTYPE(data-set-id, var-num)</td>
<td>D: VTYPE=VARTYPE(DSID,V_NAME);&lt;br&gt;M: Use %SYSFUNC</td>
</tr>
<tr>
<td>VARFMT</td>
<td>Return the corresponding variable attribute given the variable’s position in a data set.</td>
<td>VARFMT(data-set-id, var-num)</td>
<td>D: VTYPE=VARFMT(DSID,V_NAME);&lt;br&gt;M: Use %SYSFUNC</td>
</tr>
<tr>
<td>VARLEN</td>
<td>Return the corresponding variable attribute given the variable’s position in a data set.</td>
<td>VARLEN(data-set-id, var-name)</td>
<td>D: VTYPE=VARLEN(DSID,V_NAME);&lt;br&gt;M: Use %SYSFUNC</td>
</tr>
<tr>
<td>ATTRC</td>
<td>Retrieve the value from the data set descriptor information for the specified attribute.</td>
<td>ATTRC(data-set-id, attribute-name)</td>
<td>D: NOBS=ATTRC(DSID,'NOBS');&lt;br&gt;M: %LET NOBS=%SYSFUNC(ATTRC(DSID,NOBS));</td>
</tr>
<tr>
<td>ATTRN</td>
<td>Retrieve the value from the data set descriptor information for the specified attribute.</td>
<td>ATTRN(data-set-id, attribute-name)</td>
<td>D: NOBS=ATTRN(DSID,'NOBS');&lt;br&gt;M: %LET NOBS=%SYSFUNC(ATTRN(DSID,NOBS));</td>
</tr>
</tbody>
</table>

Two commonly used attributes are NOBS and N VARS which return the number of observations and variables in a data set respectively. You can use these values to control iterations over observations for “fetching” and over variables for “getting”. Note that NOBS does not take into account any WHERE clauses that may be in effect.

If there was a problem opening the data set, the function will return a value of 0. The data set name may include any data set options with the exclusion of OBS and NOBS. When the OPEN function is called in regular SAS code, the data set name should be provided as a character string or a variable name that contains a character string referencing the data set. When used in Macro code, you can pass the data set name directly without enclosing it in quotes. If no data set name is provided, the call will attempt to open the _LAST_ data set.

When a data set is first opened, and before any observations are retrieved, the next observation points to the first available observation in the data set. When an observation is retrieved using either the FETCH or FETCHOBS function, a subsequent call to the FETCH function will return the next observation following the last observation retrieved. Calls to the FETCH and FETCHOBS functions return numeric values that can be used to determine the outcome of the call. If an observation was successfully loaded into the DDV, both functions will return a value of 0. A value of -1 indicates that the end of the data set has been reached or, in the case of FETCHOBS that the observation number specified in the call exceeds the number of observations in the data set. Any other value indicates that a problem was encountered when fetching the observation.

Note that NOBS does not take into account any WHERE clauses that may be in effect.
REFERENCES:

ACKNOWLEDGMENTS
SAS is a Registered Trademark of the SAS Institute, Inc. of Cary, North Carolina.

CONTACT INFORMATION
Your comments and questions are valued and encouraged. Contact the author at:

Felix Galbis-Reig
George Washington University Biostatistics Center
6110 Executive Blvd. Suite 750
Rockville MD ZIP
Email: fgalbis@biostat.bsc.gwu.edu