Chapter III

Design of the Run-time Organization

This chapter describes the run-time organization for the translator-generated programs. The topics of concern include the representation and the storage-allocation strategies of data objects, the run-time structure of AWK subroutines, and the run-time tables.

3.1 Storage Classes for AWK Expressions

In order to accommodate various forms of run-time data storage, the run-time organization for the generated program provides five different storage classes for AWK expressions. Table 3.1 summarizes the storage classes. Each class is internally identified by a unique, small integer defined as a C preprocessor symbolic name, and is implemented by a C data type.

Table 3.1 Storage Classes for AWK Expressions

Storage Class	C Preprocessor Symbolic Name	Corresponding C Data Type
numeric	A_NUM	AWKFLOAT
nonfreeable-string	A_STR	char *
freeable-string	A_STRF	char *
variable	A_VAR	VarType *
function	A_FUNC	FnType *

Numeric Class. The expression of this class has a strict numeric value. It is implemented in C by the type AWKFLOAT, which is a C preprocessor symbol defined in awclib.h to be either double or float depending on the precision required.

Nonfreeable-string Class. The expression of this class has a strict string value, implemented in C as a character pointer. The

string value of this class cannot be deallocated (freed) after use, because it may be currently in use elsewhere, or does not come from the dynamic allocation using the standard C library. Also, operators or functions that use a string of this class normally treat it as a read-only value.

Freeable-string Class. This class is similar to its nonfreeable counterpart described above, except that the string could be, and should be, deallocated after use so as to keep the run-time dynamic space from exhaustion.

Variable Class. This class is associated with AWK variables, both built-in and user-defined, and is implemented in C as a pointer to a structure *VarType* described in section 3.2.1.

Function Class. This class is associated with the returned value from a user-defined function and is implemented in C as a pointer to a structure *FnType* described in section 3.3.1. The expression value of this class is temporary by nature, so its data structure, the FnType node, will normally be deallocated immediately after use.

Note that either numeric or string value could be obtained from an expression of variable or function class by calling some special libawc accessing routines. In fact, these two multi-typed storage classes are designed to implement an important AWK characteristic that an AWK variable and the returned value from a function call could be treated as either a number or a string depending on the context.

3.2 Implementation of AWK Variables

3.2.1 Data Structure

AWK variables are multifaceted. It has a scalar value that is either a string or a number, or both. Since there is no explicit declaration in AWK, the type of a variable must be inferred from the

context. The fact that both numeric and string value can be obtained from any AWK variable implies automatic type conversion between string and numeric value. Moreover, an AWK variable can be either a scalar variable or an array, also depending on context. Consequently, the data structure for AWK variables must be designed to accommodate all these flexibilities.

The data structure for AWK variables is a C structure called *VarType*. Each AWK variable, both built-in and user-defined, has associated with it a VarType node. There are four fields in the structure as summarized in Table 3.2.

Table 3.2 Fields in the VarType Structure

Field Name	C Data Type	Description
type	int	variable type
nval	AWKFLOAT	numeric value of the variable
sval	char *	string value of the variable
aval	ArrElem **	pointer to a structure for an associative array

Table 3.3 Variable Type

Symbolic name	nval value	sval value
V_UNDEF	undefined	undefined
V_NUM	defined	undefined
V_STR	undefined	defined
V_BOTH_NUM	defined	defined
V_BOTH_STR	defined	defined

Table 3.2 shows that the structure VarType have three value fields *nval*, *sval*, and *aval*, to store three different types of values. The field *nval* stores the variable's numeric value while *sval* stores its string one. If the variable is used as an associative array, the field aval will store a pointer to the data structure for that array; otherwise this field will be NULL. The fields *nval* and *sval* are interrelated in the sense that numeric-to-string conversion of nval's value would yield the same string as the one stored in

sval, and vice versa. The field *aval*, however, is totally independent of the other two value fields.

The field *type* is used to indicate the current internal status of the variable concerning its scalar values in the fields nval and *sval*. Table 3.3 shows the five different possible type values for this field. Each type reflects whether the fields *nval* and *sval* currently contain a defined or undefined (garbage) value. The type V_UNDEF indicates that the variable's scalar values are undefined. A variable of type V_NUM currently stores a numeric value only, while that of type V_STR stores a string value only. The types V_BOTH_NUM or V_BOTH_STR indicates that both nval and *sval* fields currently have values but the variable's primary type is numeric or string, respectively. Whether a variable's primary type is numeric or string depends on whether the assignment of its value is, by nature, a numeric or a string one. The primary type of a V_NUM or V_BOTH_NUM variable is numeric, while that of a V_STR or V_BOTH_STR is string. A V_NUM variable is automatically promoted to a V_BOTH_NUM one when its string value is called for, forcing a numeric-to-string conversion of its *nval*'s value and storing the conversion result in the *sval* field. Similarly, a V_STR variable is automatically promoted to a V_BOTH_STR one in the anologous manner.

Note that the field *type* relates only to the scalar value of a variable and has nothing to do with the aval field.

3.2.2 Classification of AWK Variables

3.2.2.1 Global and Local Variables

Within a user-defined function, all the parameters named in the function's parameter list are *local* variables, lasting only as long as the function is executing, and unrelated to variables of the same name elsewhere in the program. But all other variables are *global*; that is, it is visible and accessible throughout the program.

3.2.2.2 Scalar and Array Variables

An AWK variable is a *scalar* one if it stores a numeric value or a string, or both at the same time. As described previously, the fields nval and sval of its associated VarType node will hold its numeric and string value respectively.

An AWK variable is an *array* if it is used in the program as such; for example, it appears with a subscript or is used as an argument to a function that uses the argument as an array. The VarType node associated with an array variable will have its aval field contain a pointer to a structure for an associative array.

It should be noted that the run-time organization uses the same VarType structure for both scalar and array variable. The effect of this is that each AWK variable could have dual roles of being both scalar and array at the same time if it has values both in one of its scalar fields and in its aval field simultaneously. Nevertheless, the two roles are totally independent of each other in the sense that its scalar value does not relate to and have no effect whatsoever on its array properties, and vice versa.

As a consequence, when used as global variables, a scalar variable and an array variable with the same name will behave as if they were two separate, unrelated variables even though they share the same VarType structure; but when the variable name is used as an argument to a function, both its scalar and array aspects are passed together to the function because what is passed is the shared VarType node.

3.2.3 Implementing Global Scalar Variables

3.2.3.1 Space Allocation

Each AWK global scalar variable, either built-in or user-defined, is translated into an external VarType variable in the generated program. Being a C external object ensures automatic space

allocation at run time. Each of these VarType variables is declared with the name derived from its corresponding AWK variable by appending the suffix *__AWK* to its name. For example, the built-in variable *NR* is implemented in the generated program as a VarType variable named *NR__AWK*.

The field variables are translated a little differently, however. They are declared as an external array, named *field*, of VarType structures:

```
Vartype field[MAXFIELD+1]
```

where MAXFIELD is a symbolic constant defined in awclib.h to be the maximum number of fields allowed. The variable field[0] corresponds to AWK's $0, field[1] to $1, and so on.

3.2.3.2 Initialization

Each VarType node that corresponds to a global user-defined or field variable is initialized to have the following values in its fields:

field	initial value
type	V_UNDEF
nval	(undefined)
sval	(undefined)
aval	NULL

The AWK language requires that a number of built-in variables be initialized to default values. Thus, its corresponding VarType structure is initialized accordingly. Table 3.4 shows the initial values of the fields in the VarType structures for all built-in variables.

Table 3.4 Initial Values of the VarType nodes for Built-in Variables

Variable	type	nval	sval	aval
ARGC	V_UNDEF	undefined	undefined	NULL
ARGV	V_UNDEF	undefined	undefined	NULL
ENVIRON	V_UNDEF	undefined	undefined	NULL
FILENAME	V_STR	undefined	"-"	NULL
FNR	V_NUM	0.0	undefined	NULL
FS	V_STR	undefined	" "	NULL
NF	V_NUM	0.0	undefined	NULL
NR	V_NUM	0.0	undefined	NULL
OFMT	V_STR	undefined	"%.6g"	NULL
OFS	V_STR	undefined	" "	NULL
ORS	V_STR	undefined	"\n"	NULL
RLENGTH	V_UNDEF	undefined	undefined	NULL
RS	V_STR	undefined	"\n"	NULL
RSTART	V_UNDEF	undefined	undefined	NULL
SUBSEP	V_STR	undefined	"\034"	NULL

3.2.3.3 Accessing Variable's Values

In AWK programmers' view, a variable always have both string and numeric values at the same time. Internally, however, its associated VarType node may or may not have a defined value in its nval or sval field at any given instant. In order to bridge the gap between the logical and the internal view, the values in the VarType node are normally obtained not by accessing the fields directly but by calling a set of libawc routines, namely *varNval()*, *varIval()*, *varLval()*, and *varSval()*, to get the variable's values of type AWKFLOAT, integer, long, and string, respectively. If necessary, these routines will automatically perform type conversion, store the conversion result in the appropriate field, and then update the VarType node's scalar status in the type field.

3.2.3.4 Reinitializing Variables

When a variable needs to clear its values, the associated VarType node will be reset to the undefined status. The libawc routine *renewVar()* will be called to do the job. It will free the string in the sval field if it exists and set the type field to V_UNDEF. Note that renewVar() concerns only the scalar aspect of the variable, so the aval field is left untouched.

3.2.4 Implementing Global Array Variables

The characteristic that set AWK arrays apart from those in most other languages is that subscripts are strings. This gives AWK a capability like the associative memory of SNOBOL4 tables, and for this reason, arrays in AWK are called *associative* arrays. Thus, the implementation of AWK array variables in the run-time organization is designed to serve this capability.

3.2.4.1 Space Allocation and Initialization

Since an AWK variable that is used as an array shares the same VarType node with a scalar-valued variable that has the same name, the issues of C declarations and run-time space allocation for its associated VarType node are just exactly the same as those for scalar variables. In other words, a single C declaration in the generated program:

```
VarType x__AWK;
```

will serve the uses of an AWK variable x not only as a scalar variable, such as in the statement *x=5*, but also as an array variable, such as in the statement *x["apple"]=10*.

Initially, all the VarType nodes have its aval field initialized to NULL and the data structure for an associative array to be pointed to by the aval field will not be allocated until the variable is first referenced as an array at run time by calling the

libawc routine array(). If the routine finds that the aval field of the VarType node passed to it is NULL, it will automatically allocate an associative array data structure and set the aval field to point to it.

3.2.4.2 Data Structure for AWK Arrays

An AWK array element, which appears in program with a subscript, and a scalar variable are semantically of the same storage class; thereby they could be used in the same context. The only difference is that an array element needs a unique subscript to reference itself while a scalar variable need not. Thus, the subscript has to be incorporated into the array element's data structure.

The data structure for an AWK array element is a C structure named ArrElem. There are three fields in the structure: sub, var, and next, as summarized in Table 3.5.

Table 3.5 Fields in the ArrElem Structure

Field	C Data Type	Description
sub	char *	The identifying subscript string
var	VarType	The node for all its scalar-variable properties
next	ArrElem *	points to the next ArrElem node in the same linked list

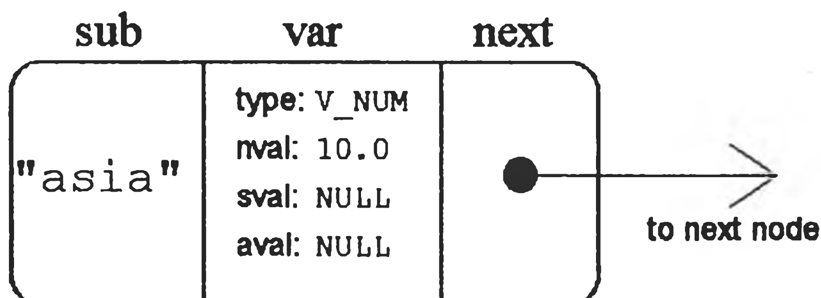

Figure 3.1 Example of an ArrElem node

Figure 3.1 illustrates the ArrElem node for an array element x["asia"] after being assigned a numeric value 10. The *sub* field stores the subscript and the *var* field stores the assigned value in exactly the same way as does a VarType node for a scalar variable.

The whole array itself is implemented as a search table of all its elements, using the hashing scheme that resolves hash collisions by separate chaining. This hash table is actually the structure that the aval field points to. Internally, it is implemented in C as an array of HASHSIZE pointers to ArrElem nodes, where HASHSIZE is a programmer-adjustable symbolic constant, defined in awclib.h, for table size. The i-th entry in the hash table is a pointer to the linked list of ArrElem nodes that share the same hash function value i. The value of the field *sub* in each ArrElem node, which is the subscript string identifying the array element, is used as the key to compute the hash function value.

Figure 3.2 illustrates the data structure for an AWK array variable x. It shows the ArrElem nodes for three array elements x["pig"], x["ox"], and x["cat"]. The subcript strings "pig" and "ox" are supposedly computed to the same hash valuc i, thus being in the same chain. The subscript "cat", however, is computed to another hash value j and is supposedly the only member of its chain.

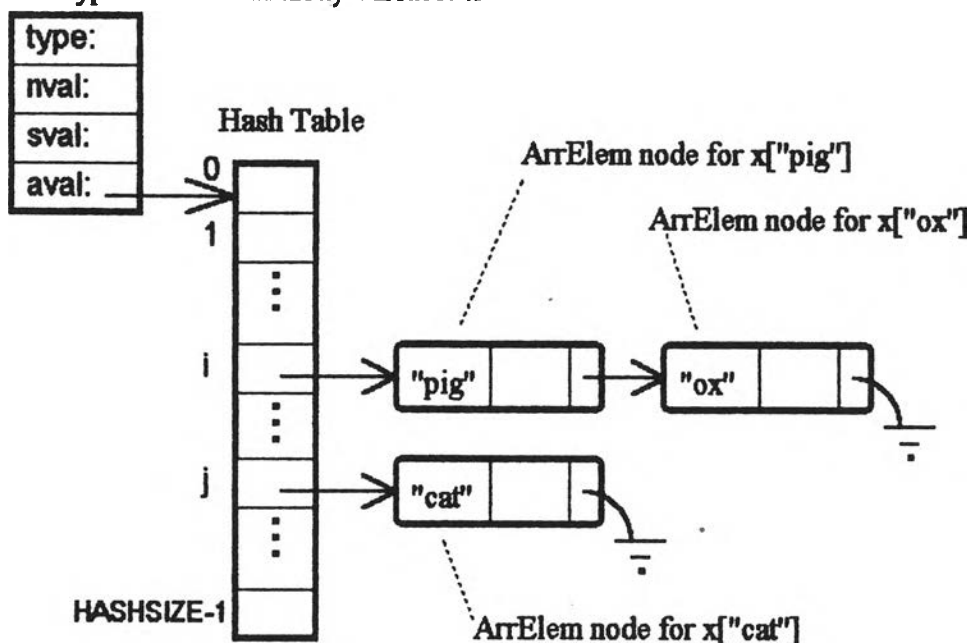

Figure 3.2 Data structure for AWK associative arrays

3.2.4.3 Accessing the Values of an Array Element

Each reference to an array element in the AWK source program is translated into a call to the libawc routine *array()*. For example,

x["thai"]

is translated into

array(&x__AWK, A_STR, "fox"),

where x__AWK is the VarType node for x. The second argument to array() specifies the storage class of the third argument, the subscript string.

The routine array() will search the hash table associated with x for the ArrElem node that has the sub field containing "fox". If such node does not exist, it will be allocated at once. Then, array() will return a pointer to the var field of that node to the caller. This means the returned value of array() has the storage class A_VAR so accessing its stored values could be done in exactly the same way as normal scalar variables.

3.2.5 Implementing Local Variables

AWK local variables have the same run-time storage class as global ones, thereby each having a unique VarType node allocated for it. However, they are different from their global counterparts in that their associated VarType nodes are defined in the generated program as automatic variables rather than external ones and are declared inside the generated C functions corresponding to AWK user-defined functions. Being of automatic storage class ensures automatic space allocation and deallocation upon function entry and exit. Also, at the beginning of each generated function, the translator has to generate codes to assign the values passed from the caller's actual arguments to their corresponding local VarType nodes in order. Except these two different attributes - C storage class and the way their values are initialized - there are no differences in use between the VarType nodes for local variables and those for global ones.

3.3 Implementation of AWK User-defined Functions

3.3.1 Function Class and The FnType Structure

As mentioned in section 3.1, a user-defined function call, or equivalently, the returned value from a user-defined function, has the storage class *function*, which is implemented in C as a pointer to the structure *FnType*. The FnType structure has two fields: type and u. The field u is a C union with four members, namely nval, sval, vval, and fval, to store the value of the class *numeric*, *freeable-string* or *nonfreeable-string*, *variable*, and *function*, respectively. The field type indicates which class of value is currently stored in the u field.

3.3.2 Function Invocation

Each AWK user-defined function is translated into a unique C function returning a pointer to a FnType node. Each run-time invocation of the generated C function explicitly allocates a new FnType node to store the invocation's logical returned value. Internally, the generated C function actually returns a pointer to

the FnType node associated with the current invocation and the caller must extract for itself the invocation's value from the FnType node. Also, the caller is responsible for deallocating the FnType node immediately after its value has been obtained.

3.3.3 Parameter Passing Mechanism

As mentioned previously, each formal parameter of an AWK user-defined function is a local variable. Implementation of local variables in the generated program is described in section 3.2.5 above. Since in AWK language the number of actual parameters passed to the function may be varied with each call, each run-time invocation of the function passes the number of passed parameters as the first parameter, followed by a sequence of actual parameter values. Each parameter value is actually passed in the form of an ordered pair of its storage class and its value; for example, the numeric value 4.5 is passed with the ordered pair (A_NUM, 4.5) and the variable x is passed with the ordered pair (A_VAR, &x__AWK), where &x__AWK is the address of the VarType node associated with x.

At the receiving end, the invoked function begins its execution by explicitly assigning the value in each passed ordered pair to the respective local variable, in order. If the number of passed ordered pairs is less than the number of local variables, then some of the local variables will remain unassigned, thereby having the status of V_UNDEF at the function beginning. Having done with the process of passing parameter values, the invoked function will then start executing its function body.

3.4 Run-time Tables

The run-time organization of the generated programs maintains two important tables, one for global variables and the other for the input/output files and pipes opened during program execution.

3.4.1 Table of Global Variables

This table serves as a mapping between each global variable name in the AWK source program and the address of its associated VarType node in the generated program. The mapping is necessary for implementing the ability to assign values to global variables at command-line. For example, an executable file *foo* that is produced from the generated program could be run by issuing the shell command:

```
foo data1 aho=5 data2
```

to specify that the numeric value 5 is to be assigned to the global variable *aho* just before reading the first record of the input file *data2*. The command-line assignment statement like this implies that the run-time organization has to know the address of the VarType node associated with the AWK variable with that name, hence the necessity of this mapping table.

Each entry in the table is a C structure *VarInfoNode*, which has three fields as summarized in Table 3.6. All the entries are allocated and installed into the table at run time.

Table 3.6 Fields in the VarInfoNode Structure

Field	C Data Type	Description
name	char *	Name of a global variable as defined in the AWK source program
pv	VarType *	Address of the associated VarType node
next	VarInfoNode *	Pointer to the next node in the linked list

3.4.2 Table of Input/Output Files and Pipes

This table maintains the information and the status of each input/output file or pipe opened during program execution. The data structure for each entry in the table is a C structure named

FileInfoNode, which has five fields as summarized in Table 3.7. The possible status values for the field *mode* are 'r', 'w', 'a', and 'c' to indicate that the file or pipe is opened for reading, writing, appending, or it has been closed, respectively.

Table 3.7 Fields in the FileInfoNode Structure

Field	C Data type	Description
name	char *	Filename or pipe command as appeared in the AWK source program
mode	int	Current status of the file/pipe
isPipe	int	1 if the entry is for a pipe, or 0 if for a file
fp	FILE *	File pointer associated with the file/pipe
next	FileInfoNode *	Pointer to the next node in the linked list