NAME

mawk-arrays – design notes for mawk's array implementation

SYNOPSIS

This is the documentation for the **mawk** implementation of awk arrays. Arrays in awk are associations of strings to awk scalar values. The **mawk** implementation stores the associations in hash tables. The hash table scheme was influenced by and is similar to the design presented in Griswold and Townsend, *The Design and Implementation of Dynamic Hashing Sets and Tables in Icon*, **Software Practice and Experience**, 23, 351-367, 1993.

DATA STRUCTURES

Array Structure

The type **ARRAY** is a pointer to a **struct array**. The *size* field is the number of elements in the table. The meaning of the other fields depends on the *type* field.

There are three types of arrays and these are distinguished by the *type* field in the structure. The types are:

AY_NULL

The array is empty and the *size* field is always zero. The other fields have no meaning.

AY_SPLIT

The array was created by the *AWK* built-in *split*. The return value from *split* is stored in the *size* field. The *ptr* field points at a vector of **CELL**s. The number of **CELL**s is the *limit* field. It is always true that $size \le limit$. The address of *A[i*] is *(CELL*)A->ptr+i-1* for $1 \le i \le size$. The *hmask* field has no meaning.

Hash Table

The array is a hash table. If the **AY_STR** bit in the *type* field is set, then the table is keyed on strings. If the **AY_INT** bit in the *type* field is set, then the table is keyed on integers. Both bits can be set, and then the two keys are consistent, i.e., look up of *A[-14*] and *A["-14"*] will return identical *CELL* pointers although the look up methods will be different. In this case, the *size* field is the number of hash nodes in the table. When insertion of a new element would cause *size* to exceed *limit*, the table grows by doubling the number of hash chains. The invariant, (*hmask*+1)*max_ave_list_length=limit* is always true. *Max_ave_list_length* is a tunable constant.

Hash Tables

The hash tables are linked lists of nodes, called **ANODE**s. The number of lists is *hmask+1* which is always a power of two. The *ptr* field points at a vector of list heads. Since there are potentially two types of lists, integer lists and strings lists, each list head is a structure, *DUAL_LINK*.

The string lists are chains connected by *slinks* and the integer lists are chains connected by *ilinks*. We sometimes refer to these lists as slists and ilists, respectively. The elements on the lists are **ANODE**s. The fields of an **ANODE** are:

slink The link field for slists. *ilink* The link field for ilists. *sval* If non-null, then *sval* is a pointer to a string key. For a given table, if the **AY_STR** bit is set then every **ANODE** has a non-null *sval* field and conversely, if **AY_STR** is not set, then every *sval* field is null.

hval The hash value of *sval*. This field has no meaning if *sval* is null.

ival The integer key. The field has no meaning if set to the constant, *NOT_AN_IVALUE*. If the **AY_STR** bit is off, then every **ANODE** will have a valid *ival* field. If the **AY_STR** bit is on, then the *ival* field may or may not be valid.

cell The data field in the hash table. \ndhitems

So the value of *A*[*expr* is stored in the *cell* field, and if *expr* is an integer, then *expr* is stored in *ival*, else it is stored in *sval*.

ARRAY OPERATIONS

The functions that operate on arrays are,

CELL array_find(ARRAY A, CELL *cp, int create_flag)*
: returns a pointer to *A*[*expr*] where *cp* is a pointer to the *CELL* holding *expr*. If the *create_flag* is on and *expr* is not an element of *A*, then the element is created with value *null*.

*void array_delete(ARRAY A, CELL *cp)*
: removes an element *A*[*expr* from the array *A*. *cp* points at the *CELL* holding *expr*.

void array_load(ARRAY A, size_t cnt)
: builds a split array. The values *A[1..cnt]* are moved into *A* from an anonymous buffer with *transfer_to_array()* which is declared in *split.h*.

void array_clear(ARRAY A) removes all elements of *A*.
: The type of *A* is then **AY_NULL**.

*STRING** array_loop_vector(ARRAY A, size_t *sizep)*
: returns a pointer to a linear vector that holds all the strings that are indices of *A*. The size of the the vector is returned indirectly in **sizep*. If *A->size*≡**0**, a *null* pointer is returned.

CELL array_cat(CELL *sp, int cnt)*
: concatenates the elements of *sp[1-cnt..0]*, with each element separated by *SUBSEP*, to compute an array index. For example, on a reference to *A*[i,j], *array_cat* computes *i ○ SUBSEP ○ j where ○ denotes concatenation.*

Array Find

Any reference to A[*expr*] creates a call to *array_find(A,cp,CREATE)* where *cp* points at the cell holding *expr*. The test, *expr* **in** *A*, creates a call to *array_find(A,cp,NO_CREATE)*.

Array_find is a hash-table lookup function that handles two cases:

1. If **cp* is numeric and integer valued, then lookup by integer value using *find_by_ival*. If **cp* is numeric, but not integer valued, then convert to string with *sprintf(CONVFMT,...)* and go to case˜2.
2. If **cp* is string valued, then lookup by string value using *find_by_sval*. \ndlist

To test whether *cp->dval* is integer, we convert to the nearest integer by rounding towards zero (done by *do_to_I*) and then cast back to double. If we get the same number we started with, then *cp->dval* is integer valued.

When we get to the function *find_by_ival*, the search has been reduced to lookup in a hash table by integer value.

When a search by integer value fails, we have to check by string value to correctly handle the case insertion by *A["123"]* and later search as *A[123]*. This string search is necessary if and only if the **AY_STR** bit is set. An important point is that all **ANODE**s get created with a valid *sval* if **AY_STR** is set, because then creation of new nodes always occurs in a call to *find_by_sval*.

Searching by string value is easier because *AWK* arrays are really string associations. If the array does not have the **AY_STR** bit set, then we have to convert the array to a dual hash table with strings which is done by the function *add_string_associations*.

One *Int* value is reserved to show that the *ival* field is invalid. This works because *d_to_I* returns a value in *[-Max_Int, Max_Int*].

On entry to *add_string_associations*, we know that the **AY_STR** bit is not set. We convert to a dual hash table, then walk all the integer lists and put each **ANODE** on a string list.

Array Delete

The execution of the statement, **delete** *A*[*expr*], creates a call to *array_delete(ARRAY A, CELL *cp)*. Depending on the type of **cp*, the call is routed to *find_by_sval* or *find_by_ival*. Each of these functions leaves its return value on the front of an *slist* or *ilist*, respectively, and then it is deleted from the front of the list. The case where *A*[*expr* is on two lists, e.g., *A[12*] and *A["12"*] is checked by examining the *sval* and *ival* fields of the returned **ANODE***.

Even though we found a node by searching an *ilist* it might also be on an *slist* and vice-versa.

When the size of a hash table drops below a certain value, it might be profitable to shrink the hash table. Currently we don't do this, because our guess is that it would be a waste of time for most *AWK* applications. However, we do convert an array to **AY_NULL** when the size goes to zero which would resize a large hash table that had been completely cleared by successive deletions.

Building an Array with Split

A simple operation is to create an array with the *AWK* primitive *split*. The code that performs *split* puts the pieces in an anonymous buffer. *array_load(A, cnt)* moves the *cnt* elements from the anonymous buffer into *A*. This is the only way an array of type **AY_SPLIT** is created.

If the array *A* is a split array and big enough then we reuse it, otherwise we need to allocate a new split array. When we allocate a block of **CELL**s for a split array, we round up to a multiple of 4.

Array Clear

The function *array_clear(ARRAY A)* converts *A* to type **AY_NULL** and frees all storage used by *A* except for the *struct array* itself. This function gets called in three contexts:

(1) when an array local to a user function goes out of scope,

(2) execution of the *AWK* statement, *delete A* and

(3) when an existing changes type or size from *split()*.

Constructor and Conversions

Arrays are always created as empty arrays of type **AY_NULL**. Global arrays are never destroyed although they can go empty or have their type change by conversion. The only constructor function is a macro.

Hash tables only get constructed by conversion. This happens in two ways. The function *make_empty_table* converts an empty array of type **AY_NULL** to an empty hash table. The number of lists in the table is a power of 2 determined by the constant *STARTING_HMASK*. The limit size of the table is determined by the constant *MAX_AVE_LIST_LENGTH* which is the largest average size of the hash lists that we are willing to tolerate before enlarging the table. When *A->size* exceeds *A->limit*, the hash table grows in size by doubling the number of lists. *A->limit* is then reset to *MAX_AVE_LIST_LENGTH* times *A->hmask+1*.

The other way a hash table gets constructed is when a split array is converted to a hash table of type **AY_INT**.

To determine the size of the table, we set the initial size to *STARTING_HMASK+1* and then double the size until $A\text{->}size \le A\text{->}limit$.

Doubling the Size of a Hash Table

The whole point of making the table size a power of two is to facilitate resizing the table. If the table size is 2^n and *h* is the hash key, then *h* **mod** 2^n is the hash chain index which can be calculated with bit-wise and, *h* & (2^{n-1}). When the table size doubles, the new bit-mask has one more bit turned on. Elements of an old hash chain whose hash value have this bit turned on get moved to a new chain. Elements with this bit turned off stay on the same chain. On average only half the old chain moves to the new chain. If the old chain is at *table*[i], $0 \le i < 2^n$, then the elements that move, all move to the new chain at *table*[i + 2^n].

As we walk an old string list with pointer *p*, the expression *p->hval & new_hmask* takes one of two values. If it is equal to *p->hval & old_hmask* (which equals *i*), then the node stays otherwise it gets moved to a new string list at *j*. The new string list preserves order so that the positions of the move-to-the-front heuristic are preserved. Nodes moving to the new list are appended at pointer *tail*. The **ANODE**s, *dummy0*˜and *dummy1*, are sentinels that remove special handling of boundary conditions.

The doubling of the integer lists is exactly the same except that *slink* is replaced by *ilink* and *hval* is replaced by *ival*.

Array Loops

Our mechanism for dealing with execution of the statement,

for (*i* in *A*) { *statements* }

is simple. We allocate a vector of *STRING** of size, *A->size*. Each element of the vector is a string key for˜*A*. Note that if the **AY_STR** bit of *A* is not set, then *A* has to be converted to a string hash table, because

the index *i* walks string indices.

To execute the loop, the only state that needs to be saved is the address of *i* and an index into the vector of string keys. Since nothing about *A* is saved as state, the user program can do anything to *A* inside the body of the loop, even *delete A*, and the loop still works. Essentially, we have traded data space (the string vector) in exchange for implementation simplicity. On a 32-bit system, each **ANODE** is 36 bytes, so the extra memory needed for the array loop is 11 more than the memory consumed by the **ANODE**s of the array. Note that the large size of the **ANODE**s is indicative of our whole design which pays data space for integer lookup speed and algorithm simplicity.

The only aspect of array loops that occurs in *array.c* is construction of the string vector. The rest of the implementation is in the file *execute.c*.

As we walk over the hash table **ANODE**s, putting each *sval* in *ret*, we need to increment each reference count. The user of the return value is responsible for these new reference counts.

Concatenating Array Indices

In *AWK*, an array expression *A[i,j]* is equivalent to the expression *A[i SUBSEP j]*, i.e., the index is the concatenation of the three elements *i*, *SUBSEP* and *j*. This is performed by the function *array_cat*. On entry, *sp* points at the top of a stack of **CELL**s. *Cnt* cells are popped off the stack and concatenated together separated by *SUBSEP* and the result is pushed back on the stack. On entry, the first multi-index is in *sp[1-cnt]* and the last is in *sp[0]*. The return value is the new stack top. (The stack is the run-time evaluation stack. This operation really has nothing to do with array structure, so logically this code belongs in *execute.c*, but remains here for historical reasons.)

We make a copy of *SUBSEP* which we can cast to string in the unlikely event the user has assigned a number to *SUBSEP*.

Set *sp* and *top* so the cells to concatenate are inclusively between *sp* and *top*.

The *total_len* is the sum of the lengths of the *cnt* strings and the *cnt-1* copies of *subsep*.

The return value is *sp* and it is already set correctly. We just need to free the strings and set the contents of *sp*.