The array type in gretl

Allin Cottrell

Second draft, June 21 2014

Since mid-June 2014 gretl CVS includes an array type. This is still experimental; some of the details described below may be subject to change, and further refinements are likely to be added. Changes from first draft: fix some typos and revise the script in section 5.

1 Introduction

A gretl array is, as you might expect, a container which can hold zero or more objects of a certain type, indexed by consecutive integers starting at 1. It is one-dimensional. This type is implemented by a quite "generic" back-end. The types of object that can be put into arrays are strings, matrices, bundles and lists; a given array can hold only one of these types.

Of gretl's "primary" types, then, neither scalars nor series are supported by the new array mechanism. There would be little point in supporting arrays of scalars as such since the matrix type already plays that role, and more flexibly. As for series, they have a special status as elements of a dataset (which is in a sense an "array of series" already) and in addition we have the list type which already functions as a sort of array for subsets of the series in a dataset.

2 Creating an array

An array can be brought into existence in any of three ways: bare declaration, assignment from `null`, or using the new `array()` function. In each case one of the specific type-words `strings`, `matrices`, `bundles` or `lists` must be used. Here are some examples:

```
# make an empty array of strings
strings S
# make an empty array of matrices
matrices M = null
# make an array with space for 4 bundles
bundles B = array(4)
```

The "bare declaration" form and the "`= null`" form have the same effect of creating an empty array, but the second can be used in contexts where bare declaration is not allowed (and it can also be used to destroy the content of an existing array and reduce it to size zero). The `array()` function expects a positive integer argument and can be used to create an array of pre-given size; in this case the elements are initialized appropriately as empty strings, null matrices, or empty bundles or lists.

3 Setting and getting elements

There are two ways to set the value of an array element: you can set a particular element using the array index, or you can append an element using the `+=` operator:

```
# first case
strings S = array(3)
S[2] = "string the second"
# alternative
matrices M = null
M += mnormal(T,k)
```

In the first method the index must (of course) be within bounds; that is, greater than zero and no greater than the current length of the array. When the second method is used it automatically extends the length of the array by 1.

To get hold of an element, the array index must be used (see also section 8):

```
# for S an array of strings
string s = S[5]
# for M an array of matrices
printf "\n%#12.5\n", M[1]
```

4 Operations on whole arrays

At present only one operation is available for arrays as a whole, namely appending. You can do, for example

```
# for S1 and S2 both arrays of strings
strings BigS = S1 + S1
# or
S1 += S2
```

In each case the result is an array of strings whose length is the sum of the lengths of `S1` and `S2`—and similarly for the other supported types.

5 Arrays as function arguments

One can now write hansl functions that take as arguments any of the array types, and any but `lists` in "pointerized" form.[1] In addition hansl functions may return any of the array types. Here is a trivial example for strings:

```
function void printstrings (strings *S)
  loop i=1..nelem(S) -q
    printf "element %d: '%s'\n", i, S[i]
  endloop
end function

function strings mkstrs (int n)
  strings S = array(n)
  loop i=1..n -q
    sprintf S[i] "member %d", i
  endloop
  return S
end function

strings Foo = mkstrs(5)
print Foo
printstrings(&Foo)
```

[1] Our thinking on how exactly to handle arrays of lists as function arguments is not yet very far advanced.

A couple of points are worth noting here. First, the `nelem()` function, which used to work for lists only, has now been generalized to give the number of elements in any of the "container" types (lists, arrays, bundles, matrices). Second, if you do "`print Foo`" for `Foo` an array, you'll see something like:

```
? print Foo
Array of strings, length 5
```

6 Arrays and bundles

As mentioned, the `bundle` type is supported by the array mechanism. In addition, arrays (of whatever type) can be put into bundles:

```
matrices M = array(8)
# set values of M[i] here...
bundle b
b.M = M
```

The mutual "packability" of bundles and arrays means that it's possible to go quite far down the rabbit-hole... users are advised not to get carried away `;-)`

7 Extensive definition?

Right now we don't have any means of defining an array "extensively"—that is, in the way that you can define a matrix via, for example,

```
matrix m = {1,2,3; 4,5,6}
```

It might be nice to be able to do, e.g.,

```
strings S = {"foo", "bar", "baz"} # sorry, no go!
```

but you can't at present. We *could* support this sort of syntax for all array types as well as matrices, but only if we're able to eliminate type-ambiguity. That would require limiting this syntax to cases such as the above, where we're assigning from a `{...}` expression to a variable of known type on the left. In other words, you couldn't do things like

```
eval 100 * {1,2,3}
```

(which works OK at present) because we'd have no idea what type of object the `{...}` expression is supposed to produce, and therefore how to handle it.

8 Fancier indexing?

At present you can only retrieve a single array element at a time. It might worth supporting ranges, as we do with matrices, so you could extract a given sub-array:

```
matrices Msub = M[4:6] # not yet
```

This would not be very difficult; we just haven't got that far yet.

9 Hansl only!

The new array types are intended for scripting use only: they have no GUI representation and they're unlikely ever to acquire one. However, it's possible to save arrays "invisibly" in the context of a GUI session, by virtue of the fact that they can be packed into bundles, and bundles can be saved as part of a "session".

And that's all for now.