A Novel Use of Fortran

Michael Metcalf

Roland Windmolders

Abstract

The introduction of the new Fortran 90 standard makes possible, for the first time with this language, the maintenance of data structures without having to resort to auxiliary packages written especially for the purpose. As an experiment, a data structuring module, eagle, has been written in Fortran 90, making extensive use of pointers, dynamic storage, data hiding, explicit interfaces and recursion. It has been tested by using it to maintain the alignment files of the SMC experiment.

(Submitted to a CHEP'94 poster session)
1 Purpose and outline of eagle

Earlier versions of Fortran have frequently been criticized for the absence of language features that enable programmers conveniently to establish and manipulate data structures. The new standard, Fortran 90 [1], contains all the required features, such as pointers, recursion and dynamic memory allocation, that are required to program data structures directly. Its data hiding capability provides further safety. These features are fully described in [2], which contains also an extended example using them as an appendix.

As an experiment, a module containing many of the facilities required to support and manipulate tree structures has been written. It is called eagle and consists of less than 800 lines of Fortran 90. This demonstrates one of the remarkable features of the language—the succinct way in which it can be used to express algorithms. As an example, the complete code to traverse a tree deallocating all its associated storage is simply:

```fortran
recursive subroutine finish (tree)
  !
  ! Traverse a complete tree or subtree, deallocating all storage
  ! (except the state variable).
  type(data), pointer :: tree
  integer loop
  !
  do loop = 1, size(tree%p)         ! loop over children
    call finish(tree%p(loop)%pp)   ! delete all their subtrees
  end do
  deallocate(tree%j, tree%y, tree%p, tree%link)
  deallocate(tree)
end subroutine finish
```

For the purposes of this paper, a tree data structure consists of a set of connected nodes, arranged in levels. The top node points to some nonzero number of nodes at the second level. All other nodes point at zero or more nodes at the next lower level, and point back to exactly one node at the next higher level. The set of nodes pointed at by a single higher node is referred to as a layer. This is thus a standard tree of mother nodes each connected to a set of daughter nodes.

The module supports an arbitrary number of independent trees. They may be manipulated simultaneously (but see note on subroutine start below).

At each node are stored, as far as the user is concerned:
- an index number supplied by the user (does not have to be sequential),
- the index of the node that the node points back to (its mother node),
- an optional fixed character component (could be made variable),
- an optional integer array,
- an optional real array,
- and the optional links (pointers) to nodes with specified index numbers (the daughter nodes).

The node contains also a running index number maintained by the module.

In a development version, an optional array of a fixed derived-data type may also be stored at each node, as well as a single reference link to another node (even in a different tree). A reference link is a pointer between any two nodes and is not part of the tree structure as such.

The user interfaces are:
start must be called to initialise a tree immediately before the first call to new_node for that tree (i.e. no reference to another tree may be made between these two calls).

new_node stores the data provided at the node whose index number is supplied as a second argument, and sets up pointers to all the specified daughter nodes that will be stored in subsequent calls. It makes some consistency checks.

retrieve retrieves a specified node; the data arrays (if present) are accessed via pointers, as are the links (pointers) to any daughter nodes.

next like retrieve, for the node with the next following running index number (not user index number).

next_in_layer like next, for the next node in the current layer. (In a sequence of calls, the next layer down will be automatically taken when the current one is exhausted, and this can be detected by a change in the value of the back pointer. If such a call follows one to next or retrieve the next layer is not taken.)

previous retrieves the data in the mother node of the last node accessed.

dump_tree write a complete tree to a specified unit.

restore_tree read a complete tree from a specified unit (initialisation is not required, even for a new tree).

set_reference establishes a reference link between a node of one tree and a node of the same or another tree.

get_reference like retrieve, but for the data at the node that is the target of the reference link at the specified node.

finish deallocate all the storage occupied by a complete tree.

These interfaces are simple. To add a new node to a tree, with a name, some integer data, and links to three daughter nodes, we write, for example,

```fortran
    call new_node(tree_name, index, node_name, &
      integer_data = (/ (i, i = 1, 10) /), links = (/ 2, 3, 5/))
```

and to retrieve data we write

```fortran
    call retrieve(tree_name, index, back, name, j, y, link)
```

where back is the index of the mother of the node index, and j, y, and link are pointers to any optional integer data, real data or links stored at that node. These names are chosen by the user, and retrieved data can be referred to directly, say as j(16).

Internally, the module allocates a node, and allocates data at that node if the corresponding argument is present in the call to new_node, based on the basic node definition:

```fortran
! Define the basic node type
    type, public :: data
    private
    integer :: index, amount(3), running_index
    character(max_char) :: header
    integer, pointer :: j(:), link(:)
    real, pointer :: y(:)
    type(ptr), pointer :: p(:)
    type(data), pointer :: back
    type(state), pointer :: own_state
  end type data
```

Note that the internal structure of the type is inaccessible outside the module.
If work on eagle should continue, future plans would be at least to add the following features: replace a node; add a node; extend a component; allow an error exit if insufficient storage is available for allocation for dynamic memory; further validation and navigation facilities; a second reference link to a given target overwrites the existing reference link—it should rather become a new one.

2 Use of eagle in an application

This section shows how the description of the SMC (NA47) [3] detector given on an alignment file can be structured with the help of the new features provided in Fortran 90.

As an example we define two trees managed by eagle, one for proportional and drift chambers the other for hodoscopes, and use derived data types for several other parts of the data.

2.1 Proportional/drift chambers

In the following layout, name stands for any detector name, such as P0D, P45, D67B, ... and Uk stands for any plane with orientation U, i.e. Y, Z or T and ordinal k.

```
I----------I
I det_list I
I----------I
I I
I I
I I I----------I
I I----> I "name" I
I I----------I
I I I
I I I
I I I I I----------I
I I----> I name.Uk I
I I I----------I
I I I
I I I
I I I I I----------I
I I----> I name.Uk.RMS I
I I I----------I
I I I
I I I
I I I I----------I
I I I----> I ST67.STPI I
I I----------I
I----------I
I "name" I
I----------I
I
I---------->
```

Index

I det_list

I detr

I ndt = ldetr*10**6 +

detector_ordinal *10**4

ln = ndt +

plane_ordinal*100

lnk = ln + 1

If work on eagle should continue, future plans would be at least to add the following features: replace a node; add a node; extend a component; allow an error exit if insufficient storage is available for allocation for dynamic memory; further validation and navigation facilities; a second reference link to a given target overwrites the existing reference link—it should rather become a new one.

2 Use of eagle in an application

This section shows how the description of the SMC (NA47) [3] detector given on an alignment file can be structured with the help of the new features provided in Fortran 90.

As an example we define two trees managed by eagle, one for proportional and drift chambers the other for hodoscopes, and use derived data types for several other parts of the data.

2.1 Proportional/drift chambers

In the following layout, name stands for any detector name, such as P0D, P45, D67B, ... and Uk stands for any plane with orientation U, i.e. Y, Z or T and ordinal k.

```
I----------I
I det_list I
I----------I
I I
I I
I I I----------I
I I----> I "name" I
I I----------I
I I I
I I I
I I I I I----------I
I I----> I name.Uk I
I I I----------I
I I I
I I I
I I I I I----------I
I I----> I name.Uk.RMS I
I I I----------I
I I I
I I I
I I I I I----------I
I I----> I ST67.STPI I
I I----------I
I----------I
I "name" I
I----------I
I
I---------->
```

Index

I det_list

I detr

I ndt = ldetr*10**6 +

detector_ordinal *10**4

ln = ndt +

plane_ordinal*100

lnk = ln + 1

If work on eagle should continue, future plans would be at least to add the following features: replace a node; add a node; extend a component; allow an error exit if insufficient storage is available for allocation for dynamic memory; further validation and navigation facilities; a second reference link to a given target overwrites the existing reference link—it should rather become a new one.
2.2 Hodoscopes

```
     I-------I
     I hodos I
     I-------I
     I I
     I I
     I I I-------I
     I I----> I BHOD I ln = lndt +
     I I-------I detector_ordinal
     I I
     I I
     I I I-------I
     I I----> I BHOD.Uk I ln = lndt +
     I I-------I plane_ordinal*100
     I I
     I I
     I I I-------I
     I I----> I BHOD.Uk.RMS I lk = ln + 1
     I I I-------I
     I I
     I I
     I I I-------I
     I I----> I BHOD.Uk.TO I lk = ln + 2
     I I-------I
     I I
     I-------> (trigger hodoscopes---not implemented yet)
```

2.3 Derived data types

Various derived data types have been defined for the description of the corresponding piece of equipment: `target` (polarised target), `magnet` (forward spectrometer magnet), `calor` (H2 calorimeter), `absor` (hadron absorber), and `mgpos` (last bending magnet in beam-line). An example is

```
type target
    real, dimension(3) :: center
    real, dimension(4) :: par
    real, dimension(2) :: dir
    integer :: year
end type target
```

2.4 Program use

For every block on the input file, the program prints a message telling whether it has been included in the structure or not. In general, blocks may be read in any order compatible with the implied logic: the list of detectors must appear before any particular detector information but data related to various chambers and hodoscopes may be interleaved.
Two special blocks related to the ST67 chambers have been included in the detector structure: the X positions (STWX) are added to the corresponding plane information (one additional number in the real data array), and the list of variable pitches (STPI) is added as an additional bank after the last plane.

Most of the blocks not treated so far can easily be connected to the existing trees: trigger hodoscopes, as shown in the previous subsection, and time calibrations and non-linearity parameters for drift chambers at name.Uk with links (ln+2), (ln+3), etc.

The main program contains several examples of how data can be found in the structures, either by using the general subroutine retrieve (included in eagle) to search for a bank with a given index value, or by using a subroutine adapted for SMC structures only and searching for a block with a given name (subroutine srname).

2.5 Beam track fitting with eagle

As an application using a structure managed by eagle, we have rewritten the procedure fitting the direction of the incident muons in the SMC experiment. Normally this procedure is included in the pattern recognition program (Phenix) and uses hits in two sets of hodoscopes (BHA and BHB) separated by a distance of 10m, and in a small proportional chamber (POB) located behind the last hodoscope. As a first step, the beam track parameters are determined by a least-squares fit using associated hits in the two groups of hodoscopes. These lines are extrapolated to POB where corresponding hits are searched for within a given roadwidth and finally a new least-squares fit, using the three detectors, is performed.

In the working version of the SMC software, the data structure containing the hit coordinates in each plane is managed by ZEBRA [4] while all parameters related to the detectors, such as X positions, wire spacing and rms, are stored under various names in a large number of different common blocks.

The present application is set up as a simple test independent of the SMC software. Hit coordinates are read in from an input file and the detector information is fetched from the two trees managed by eagle. The trees are created in the initialisation phase by reading the standard SMC alignment file. The structures a well as the labelling and numbering conventions are described in Sections 2.1 and 2.2.

The general beam fitting processor (subroutine p0bhod) and the least-squares fitting routine (subroutine stfit1) have been rewritten in Fortran 90. In both cases the number of executable statements is reduced by about 25% due to the use of the new array features, which also makes the code more readable.

The figures illustrate some aspects of the data and the results:
- Fig. 1 shows the correlation between the impacts in BHA and BHB, due to the beam phase space.
- The residuals of the fit using the beam hodoscopes only are shown in Fig. 2. Their mean is close to 0, and their rms is of the order of 0.8. The presence of some spurious associations is clearly seen.
- The difference between the hit coordinates in POB and the lines found in the hodoscopes divided by the assumed rms in POB is shown in Fig. 3. The clustersizes in POB (not taken into account) and the error on the extrapolation itself (not included in the present test) increase the width to about 3.
3 Conclusion

We have successfully written a data structuring module using new Fortran 90 features, and used it to store, retrieve and manipulate data from the SMC detector. The code of both the module and the user program is simpler and clearer than the equivalent (sic) FORTRAN 77 code and, in particular, the programmer is relieved of the burden of managing data in memory, and is able to use mnemonic names of his or her choice. The fact that the interfaces to subroutines in a module are explicit means that errors in calling sequences are detected at compile time rather than giving rise to obscure run-time bugs.

The code has incidentally been used to test the five existing Fortran 90 compilers: NAG, DEC, IBM, EPC, and Cray.

Future work will depend on other developments in the use of Fortran 90 for high-energy physics.

REFERENCES

$\mu = -0.16$
$\sigma = 3.3$

Fig. 3