Implementing the Standards...

including Fortran 2003

Malcolm Cohen

The Numerical Algorithms Group Ltd., Oxford

Nihon Numerical Algorithms Group KK, Tokyo

Contents

1. Fortran 90

2. Fortran 95

3. The Technical Reports:
   - allocatable components etc.
   - IEEE arithmetic

4. Fortran 2003

Fortran 90

“I hear you think you can write a Fortran 90 compiler.”

- about 18 months;
- correctness;
- no extensions – Standard Fortran only!
  (that didn’t last long...);
- modular design – infrastructure;
- error detection, error detection, error detection;
- “I could do with some help on the runtime library...” !

Fortran 90 Scopes

- One of the (many) moaned-about new-fangled features.
- Basically just like all the usual block-structured suspects...
- ...except that F90 is not a single-pass language
(both internal procedures and module procedures).
- Quite tricky to do properly in a single pass with backpatching
(source of bugs).
- Tricky situations often involve not detecting errors too soon.

Tricky Fortran 90 Scopes

```
MODULE m
   REAL :: x(3) = (/ 1.5, 2.5, 3.5 /)
CONTAINS
   SUBROUTINE s
      CALL inner
   CONTAINS
      SUBROUTINE inner
         PRINT *,x(7)    ! This is not an error!
      END SUBROUTINE
      FUNCTION x(n)
         x = REAL(n)**n
      END FUNCTION
   END SUBROUTINE
END MODULE
```

Fortran 90 Modules

- Probably the most-moaned-about newfangled feature.
- Nearly trivial to implement (easier than nested scopes).
- "Lazy" lookup with caching...
- ...can be more than 1000 (!) times faster than non-lazy.
- Vendors who don't know about lazy techniques think that slow "compile time is proper".
- Module file format is text: nearly human-readable.

Lazy Module Handling

`USE module,local_1=>remote_1, local_2=>remote_2`

- Stored essentially as that.
- Module symbols *not* imported into the local symbol table.
- Referenced module symbols have local `S_REFERENCE` symbols under the local name (caching).
- ...million-symbol modules are fast.
- ...deep module use trees are fast.
- (Only referenced symbols checked for name clash.)

Fortran 90 Array Features

1.0a

- Correctness and portability > efficiency.
- Only did unary and binary array operations.
- When in doubt, make an array temp.
- A = B + C*D becomes
  aTmp1 = C*D; aTmp2 = B+aTmp1; a = aTmp2
- All intrinsics by procedure call (even SIZE et al).
- WHERE construct a scary elemental affair (close your eyes and hope). Mask is “array master”.
- Result: good answers but slow.

1.2

- The best of the 1.x releases.
- Stuck with basic design, no “performance” rewrites.

Fortran 90 Array Features, 2

2.0 • Rewrote all array features to do scary elementalisation:

- ...far fewer array temps;
- ...much **much** faster than 1.x;
- ...much **much** buggier than 1.x.
- It got the right answers in all the simple cases, but the complicated ones... were complicated.

2.2 • Serial HPF, including the `HPF_LIBRARY` module.

- The first 2.x release to be as stable as 1.2.
- Basic `FORALL` statement and construct...
- ...ugh. But worse was to come...

Fortran 95

- A minor revision, but more changes than expected.
- Some "trivial" changes were secret...
- Release 3.0 of NAGWare f90.
- Renamed to be release 1.0 of NAGWare f95 by marketting.
- Also renamed the header file, various messages, ...
- ...but not the modules (e.g. `F90_KIND`, `F90_IOSTAT`), because of backwards compatibility.

The Semantics of FORALL

```
FORALL (i=1:n,maskfun(i))
  a(i) = b(i) + c(i)
  ix(i) = iy(i) + iz(i)
END FORALL
```

is defined as

```
ArrayTemp LOGICAL mtmp;
ArrayTemp REAL rhstmp1; ArrayTemp INTEGER rhstmp2
ALLOCATE (mtmp(n),rhstmp1(n))
FORALL (i=1:n) mtmp(i) = maskfun(i)
FORALL (i=1:n) IF (mtmp(i)) rhstmp1(i) = b(i) + c(i)
FORALL (i=1:n) IF (mtmp(i)) a(i) = rhstmp1(i)
DEALLOCATE(rhstmp1); ALLOCATE(rhstmp2(n))
FORALL (i=1:n) IF (mtmp(i)) rhstmp2(i) = d(i) + e(i)
FORALL (i=1:n) IF (mtmp(i)) d(i) = rhstmp2(i)
```

The Irony of FORALL

- "world's slowest high-performance feature" (HPC vendor 1)
- "months just to get the semantics right..." (HPC vendor 2)
- "even on massively parallel machines, it's slower than DO" (HPC vendor 3)
- The analysis needed to eliminate the costly array temps... ...would parallelise the obvious `DO` loop alternative.

So at best `FORALL` is as good as `DO`, usually it is slower, sometimes *much* slower.

The Horror of FORALL

```
REAL,POINTER :: x(:,:,:), y(:,:,:), z(:,:,:)
...
FORALL (i=1:n,j=1:m,maskfun(i,j))
  x(i,i:j,j) = cos(y(i,i:j,m-j)) + sin(z(i,m-j:n-i,j))
END FORALL
```

Must evaluate rhs over entire iteration space before assignment.

But in each iteration, the rhs is a different length vector!

Implementing Horror

An arbitrary ragged-shape temporary $\rightarrow$ a list of array temps.

1. In each iteration: allocate the right size of array temp,
   evaluate the rhs,
   append the array temp to the list.

2. Re-iterate: take the first array temp from the list,
   assign it to the lhs,
   and deallocate it.

3. In fused assignments, multiple array temp lists may be in use.

Technical Reports - Allocatable

- Allocatable components, dummy arguments, functions.
- Old style allocatable arrays were a separate address and "Info" record.
- "New style" allocatable array representation is a single struct to allow efficient selection as a component and passing as an actual argument.
- Backwards compatibility with pre-compiled modules.
- Due to C and/or O.S. limitations, most arrays end up on the heap, so auto deallocation at the end of the routine is slower than it could be (still quite fast).

Technical Reports - IEEE arithmetic

- “Intrinsic” module.
- Was originally envisaged as user-suppliable, but...
- ...due to higher ambitions, ended up as necessarily built-in.
- Better than raw IEEE, or C99 (faint praise, but NDI).
- Modes flow down, flags fly up; thus...
- ...easy to understand;
- ...natural preservation of existing performance.

IEEE module implementation

- Only if `IEEE_GET_FLAG` is directly called in a routine:
  save then clear the flags on entry,
  merge the flags on exit.

- Only in a routine that uses a mode setting procedure:
  save mode on entry,
  restore mode on exit.

- Parallelism and other optimisations are little impeded by the use of IEEE facilities (all IEEE semantics being local).

Fortran 2003 - Overview

- Major revision.
- Many data enhancements.
- Many i/o enhancements.
- Initialization expressions can invoke any intrinsic function.
- Many other enhancements.
- Far too big to add to an existing compiler in one step.
- NAG is taking about 5 steps... we are about halfway...

F2003 Data Enhancements

- Type extension and polymorphism (object orientation):
  - single inheritance;
  - almost completely type-safe;
  - `SELECT TYPE` construct;
  - type-bound procedures for dynamic dispatch;
  - object-bound procedures for even more dynamic dispatch!

- Parameterised derived types (1988's problem/answer).

- Deferred character length, scalar allocatables.

- Et cetera.

F2003 Derived Type Headers

- Polymorphic object "signature" is a **type header** pointer; signatures are needed for `SELECT TYPE` (type testing).

- Type testing can be done in constant time:
  - store the type depth in the header;
  - store *all* the ancestor type-links in the header, indexed backwards from the base address;
  - (forwards from the base address is the dispatch table);
  - test: the depth value and only one back-link;
  - overhead is very small, and per type (not per object).

- Polymorphic arrays are homogenous, so signature overhead is one pointer per user object, whether array or scalar.

F2003 I/O Enhancements

- Recursive i/o! (Only with internal files.)

- Stream files. (Both text and binary.)

- Extra i/o options: `DECIMAL=` (and `DC`, `DP` edit descriptors), `IOMSG=`, `ENCODING=`, `ASYNCHRONOUS=`.

- Systematic i/o options: `SIGN=` on `OPEN/WRITE/INQUIRE`, `BLANK=` and `PAD=` on `READ`, `DELIM=` on `WRITE`.

- Two ways of telling whether an `IOSTAT` value is end of file (vs. end of record).

More F2003 I/O Enhancements

- Standardised forms for IEEE ∞s and NaNs; ability to read, not just write.

- List-directed output incompatible (real zero = F format, previously E format).

- Astounding `ROUNDING=` (and `RU`, `RD` et al edit descriptors). Does any vendor realise the wording of the standard appears to require exact i/o conversions?
(With IEEE quad precision, we are talking 65536 decimal digit arithmetic...!)

Implementing F2003 I/O enhancements

1. These all point to a complete rewrite from the ground up.

2. Backwards compatibility makes this a little less straight-forward.

3. Limiting I/O recursion to internal files makes no sense whatsoever; we have to do all the work anyway, we might as well allow it for external files too.

Traditional I/O Implementation

- Co-routine structure.
  1. Establish format.
  2. Format processing in the i/o library... ...handing back to the user program on data edit descriptors.
  3. User program calls the i/o library for each i/o list item.

- Unformatted i/o has a similar structure; allows very long i/o records without overly large buffers.

F2003 I/O Implementation Details

- I/O context structure.

- I/O context stored in the compiled program,
  not in the runtime library.

- Opaque, bigger than minimum to allow for future expansion.
  (Storing it in the user program rather than the runtime library makes additional space imperative.)

- I/O context includes all formatting information,
  so is quite large...

And after F2003... the future?

- NAG will probably be first full F2003 compiler, but...
- ...we won't reach that until 2008.
- So Fortran 2008 had better be a small update, or...
- ...further away (like, not next year!).
- Currently the standard is trying to do a big update in 2008, before we even have *any* F2003 compilers.
- Risks becoming irrelevant to real users/vendors – a mere wishlist (plan for vapourware).

That's all folks!

These slides will be available on our website.

`http://www.nag.co.uk/`