Experimental Programming in Fortran

Arjen Markus

Deltares

July 3, 2020

Introduction

Fortran programming often very pragmatic:
A concrete problem needs to be solved.

A typical program:

- Read input
- Do the calculation
- Write results

But let us get beyond the pragmatic!

More than meets the eye

Other languages offer:

- Lambda expressions, that is, anonymous functions
- Functional programming
- Non-class based object-oriented programming
- ...

And what about mathematical subjects?

Can we have some of these? That is the subject of this talk.

Mathematical notions

Consider the mathematical concept of *vector spaces*:

- Two objects (in the mathematical sense) can be added to give a new object.
- You can scale an object by a scalar.

Vectors in n-dimensional Euclidean space (R^n) are easy – but how about function spaces?

Yes, we can!

Mathematical notions (2)

```
program test_space
    use vectors_function
    implicit none
    intrinsic :: sin, cos

    type(vector_function)      :: a, b, c, d
    procedure(f_of_x), pointer :: f

    f => sin; call setfunc( a, f )  ! procedure pointer
    f => cos; call setfunc( b, f )  ! needed

    c = a + b
    d = 10.0 * c

    write(*,*) 'a at x = 1.0: ', a%eval(1.0)
    write(*,*) 'b at x = 1.0: ', b%eval(1.0)
    write(*,*) 'c at x = 1.0: ', c%eval(1.0)
    write(*,*) 'd at x = 1.0: ', d%eval(1.0)
    ...
end program test_space
```

Mathematical notions (3)

And the output is as expected:

```
$ ./test_space
 a at x = 1.0:   0.841470957
 b at x = 1.0:   0.540302277
 c at x = 1.0:    1.38177323
 d at x = 1.0:    13.8177319
```

Perhaps not a very practical example, but it illustrates that such things are possible!

Mathematical notation

Consider this advection-diffusion-reaction equation:

$$\frac{\partial C}{\partial t} + \underline{u} \cdot \nabla C = \nabla(D \nabla C) - k(C) \tag{1}$$

$$k(C) = k_0 \quad \text{if} \quad C > 0, \quad \text{else} \quad 0 \tag{2}$$

An alternative notation (using $\nabla \cdot \underline{u} = 0$, as the flow field is conservative):

$$\frac{\partial C}{\partial t} + \mathrm{div}(\underline{u} C) = \mathrm{div}(D \,\mathrm{grad}\, C) - k(C) \tag{3}$$

Mathematical notation (2)

Numerous numerical methods are available:

- The concentration C is approximated via some discretisation – a regular grid or a triangular mesh or ...
- The gradient terms are approximated using finite differences or finite-element techniques
- Discrete time steps

But: we can hide all those gruesome details!

Mathematical notation (3)

Define suitable overloaded and user-defined operations to arrive at:

```
decay = merge( decay0, 0.0, conc > 0.0 )

deriv = .div. (-flow * conc + disp * .grad. conc) - decay

conc = conc + deriv * deltt
```

Note that the code is independent of the actual discretisation and the number of dimensions.
We can hide the precise numerical methods inside the overloaded operations and functions. *(And use coarrays underneath.)*

Something completely different: lambda expressions

Languages like Java and C# allow the user to define anonymous functions, also known as *lambda expressions*.

Here is an example in Java:

```
printPersons(
    roster,
    (Person p) ->
      p.getGender() == Person.Sex.MALE
      && p.getAge() >= 18
      && p.getAge() <= 25
);
```

Here `(Person p) -> ...` defines an expression that acts as a function inside `printPersons`.

This is not possible in Fortran – or is it?

Lambda expressions – in Fortran

Well, perhaps not as elegant, but we can get close – without special syntax:

```
program print_table
    use lambda_expressions

    type(lambda_integer)               :: x
    type(lambda_expression)            :: lambda1, lambda2
    integer                            :: v

    call lambda1%set( x, x+2 )
    call lambda2%set( x, x*2 )

    do v = 1,10
        write(*,*) v, lambda1%eval(v), lambda2%eval(v)
    enddo
end program
```

Lambda expressions – overloaded operations

The secret lies in overloading the arithmetic operations and elementary functions:

```
function integer_add( x, y ) result(add)
    type(lambda_integer), intent(in), target  :: x, y
    type(lambda_integer), pointer             :: add

    allocate( add )

    add%operation =  1 ! Indicates addition
    add%first     => x
    add%second    => y
end function integer_add
```

This way, you build up an expression tree

Lambda expressions – dirty work behind the screens

And some messing about with pointers:

```
subroutine set_expression( lambda, x, expr )
    class(lambda_expression)        :: lambda
    type(lambda_integer), target    :: x
    type(lambda_integer), pointer   :: expr

    type(lambda_integer_pointer), dimension(size(lambda%operand)) :: &
                                    arg

    arg(1)%arg => x
    arg(2)%arg => null() ! Only one variable in the expression
    ...
    !
    ! Correct the pointers to arguments
    !
    call correct_pointer( arg, lambda%operand, expr )

    allocate( lambda%expr, source=expr )
end subroutine set_expression
```

Lambda expressions useful?

The fact that other languages have lambda expressions (the idea originated in LISP, if I am not mistaken) could be an argument to include them in some future version of the standard.

Having a basic implementation, however, allows us to experiment and see if it is worth the trouble.

But I really like the fact that for this experiment we do not need new syntax!

Alternative object-oriented paradigms

The Fortran 2003 standard introduced *object-oriented programming*, based on *classes* – the C++ style.

Typical concepts:

- Objects based on *classes*, extensible via *inheritance*
- OOP allows for *data abstraction* and *information hiding* (or implementation hiding)
- Objects can act as different types (*polymorphism*)

But there are other styles as well!

Alternative object-oriented paradigms: Self

The Self language for instance uses a "prototype" approach:

- Objects can be extended with new properties and methods, so no fixed classes
- Objects can be used as a template for other objects
- As a consequence: a dynamic system

A simple example of this approach in Fortran:

```
use prototypes
type(prototype) :: p1, p2
integer         :: start, end
logical         :: found
!
! Fill properties for variable p1
!
call prototype_set( p1, "Start", 1 )
call prototype_set( p1, "End", 10 )
!
! We need a copy, reset one property:
!
p2 = p1         ! This relies on automatic reallocation!

call property_set( p2, "Start", 2 )
```

Copying an object results in a new object that can get new values for existing properties or get new properties and methods.

Alternative object-oriented paradigms: implementation aspects

The implementation of the `prototypes` module depends on several Fortran 2003 features:

- Automatic and sourced allocation
- Unlimited polymorphic variables
- Procedure pointers

One "problem": it uses subroutines to retrieve values, not functions

```
integer :: value
logical :: found

call prototype_get( p1, "Start", value, found )
```

Conclusion

Fortran is not perfect, there is still a lot to be wished for, but:

- Do we realise all that is possible?
- What other experiments can we do?
- Which technique will proof useful in practice?

(Short articles on these experiments as well as the source code is available on `http://flibs.sf.net`)