A proposal for templates in Fortran

Arjen Markus
arjen.markus@deltares.nl

October 21, 2019

1 Introduction

In my note ”Experimenting with generic programming features” [2] I examined the possiblities of using modules and their renaming features to implement a kind of *templates.* Basically it boiled down to the following:

- Build up the source code for a module by *including* pieces of code in a new source file.
- The code pieces use a set of generic names for the derived types that are involved.
- By *using* the resulting module and mapping the names of specific derived types to these generic names, a module is created that uses these specific types, albeit under *generic* names.

While the method works, there are a number of drawbacks:

- You cannot deal with the intrinsic types in the same way, so that a workaround had to be devised (this was solved in my note by encapsulating the variable of intrinsic type in a derived type).
- The ”generic” code uses two different fragments to build up the complete module. This is awkward in use.

For the user of a such generic module creating a specific module with a specific derived type is not very difficult. Here is an example taken from the note:

```
module m_collection_real_array
    use basic_types, only: data_type => real_type, assignment(=)

    private
    public   :: collection_array, data_type, real_kind, assignment(=)

    integer, parameter :: real_kind = kind(1.0)
```

```
include 'collection_array_body.f90'

end module m_collection_real_array
```

Note it is the *generic* name, `data_type` that this module will expose, not the specific name.

Still, the method as described has its advantages as well: without introducing new syntax, it handles several requirements proposed by Haveraaen et al. [1]: type safety and renaming features.

This eperiment inspired a proposal for bringing *templates* into Fortran.

2 Proposed extension

The proposal is fairly limited in character and modelled on modules:

- A *template* is defined in much the same way as a module:

  ```
  template name_of_template
      ... definitions (for instance: a derived type data_type)
  contains
      ... implementation of routines
  end template name_of_template
  ```

- The template is used in the following way (note the use of an intrinsic type!):

  ```
  module name_of_specific_module
      use some_module_with_types
      integer, parameter :: wp = kind(1.0d0)

      use_template name-of-template, real(kind=wp) => data_type

      ... definitions
  contains
      ... other routines
  end module name_of_specific_module
  ```

The `template` construct allows the programmer to define all the specific and generic types they need and may contain `use_template` statements.

To guarantee type safety, generic types need to be explicitly defined as far as the interface is concerned, but the actual implementation is postponed. With the interface definition of such generic types the compiler can check that the type as used within the template fulfills the "contract" and when instantiating the template, the specific type can be checked against it as well (see section 4).

In all respects other than the actual implementation of generic types, it should be valid Fortran code, so that it can be inserted when used in much the same way as via the `include` directive.

The `use_template` statement causes several things to happen – at least conceptually:

- The definitions section of the template is inserted in the definitions section of the using module and the implementation section is to be inserted in the implementation section of that module.
- The rename list is used to replace *generic* names with specific names. If wanted or convenient, other names that appear in the template's code, such as the names of functions or subroutines, can also be renamed.
- The result is filled-in source code for the using module that can be processed as if no template was referred to. What is more, the renaming clause for `use_template` causes the generic names to be replaced by specific names.

3 Details of the substitution

The required ordering of statements presents a peculiar problem for any implementation of this proposal and so do the details of type names. This section is meant to elaborate the most obvious ones.

3.1 Type names

The generic data types may be replaced by the names of derived types, so that

```
type(data_type) :: data
```

becomes:

```
type(method_parameters_type) :: data
```

But in some contexts the name itself is used:

```
select type (var)
    type is (method_parameters_type)
        ...
```

This means that with intrinsic types, both `type(data_type)` and `data_type` must be replaced by the correct intrinsic type. In other words: more is required than a literal substitution.

We also need to take care of the various attributes: `allocatable`, `pointer`, `dimension(:)` and so on.

At the rename list this can be solved via:[1]

```
use_template collection_array , &
    real , dimension(:) , allocatable => data_type
```

but it requires a different approach for `select type` cosntructs.

[1]Parsing is unambiguous despite the separate words, as the left-hand side of => can only define one type and the right-hand side can contain only one word.

3.2 Placement of the "use_template" statement

The greatest flexibility and ease of use is achieved if there are no specific restrictions to the statements that can be used in a `template` construct. But this means that we need to decide what happens to `use` statements and `implicit` statements. Consider a simple example:

```
template generic_dictionary
    use hash_functions ! General library for hash codes

    implicit none

    ! The one generic data type
    !
    type, generic :: value_type
    end type value_type

    ! We may want to rename "dictionary" if several
    ! types of dictionaries are needed, but it is
    ! not necessary
    !
    type dictionary
        type(hash_code)  :: hash
        type(value_type) :: value
    end type dictionary
end template generic_dictionary

module dictionary
    use keywords

    implicit none

    use_template generic_dictionary, &
        character(len=40) => value_type
    ...
end module dictionary
```

We cannot simply substitute the definitions section into the code for the module, as that would lead to invalid placement of the `implicit none` and `use` statements.

If the `use_template` statement appears outside the definition section of a module, for instance to define specific copies of a subroutine or function, then the code in the `contains` section of the template is to be copied in. The template must be suitable for this type of use, of course, because in the end you will to have valid Fortran code.

Proposal:
An `implicit` statement within a `template` construct should be used for the

template only. A data type does not have to be defined in this case (unlike for modules and other code), but a variable, parameter or dummy argument must have a declared type.

Proposal:
Any `use` statement in a `template` construct is added to the list of `use` statements in the using code.

4 Type safety

The template code will use the generic types in expressions or assignments. The compiler needs to be able to check that the specific types that will replace the generic ones can actually be used. To this end the properties of the generic types need to be specified. The way to do this is by defining a derived type with the required data components, operations and type-bound routines. The code:

```
type, generic :: data_type
    ! No required components
contains
    generic :: operator(+)
end type data_type
```

defines a generic type that can be applied with additions (see below for more details). The keyword `generic` guarantees that the definition is understood as identifying the properties of a generic data type without ambiguity.

If the generic type is to have a label and is to be used in a printing routine, the code might look like:

```
type, generic :: data_type
    character(len=40) :: label
contains
    procedure(print_interface) :: print
end type data_type

abstract interface
    subroutine print_interface( this, lun )
        type(data_type)      :: this
        integer, intent(in) :: lun
    end subroutine print_interface
end interface
```

This interface definition, a ”contract” as it were, allows the compiler to check that the template code uses the generic type in a consistent way and to check that the specific type that is to replace the generic type upon instantiation also adheres to this contract.

4.1 Contract details

To make the use of these interface contracts more convenient, I propose a few conventions:

- An operation like addition, could be specified explicitly via an abstract interface for each and every combination of operand types that is supported, but by default `generic :: operator()+` means that an "addition" function is defined that takes two arguments of the same generic type and returns a value of that type, similarly for subtraction. multiplication and division.

  For exponentiation this is less straightforward: the intrinsic types take either an integer as the exponent or a real/complex value. I propose that the minimum requirement is that exponentiation be done with an integer exponent. Other data types for the exponent can be defined explicitly, if needed.

- A keyword `arithmetic`, next to the `type` defines an arithmetic type. This would be a shortcut to defining all ordinary arithmetic operations separately:

  ```
  type, arithmetic :: data_type
  end type data_type
  ```

  It is allowed to define additional operations and operations with other argument types, in the same way as you can do in ordinary code.

5 Reference implementation

Because this proposal can – at least in part – be implemented as a preprocessing step. it should be possible to gain experience with it via a program that is applied to the source code and is independent of any compiler. In fact, a prototype of such a program is available in my Flibs project on SourceForge – `https://sourceforge.net/p/flibs/svncode/HEAD/tree/trunk/experiments/generics/template_preproc.f90`

Note: this prototype does not yet handle the "contract" for generic types nor is it flexible enough to deal with continuation lines for instance.

References

[1] Magne Haveraaen, Järvi Jaakko, and Damian Rouson. Reflecting on generics for Fortran, 2019.

[2] Arjen Markus. Experimenting with generic programming features, 2019.

Appendix: Full example

Here is a full example of the use of the proposed template feature. It can be processed with the prototype program and gives the expected results.[2] The idea is to have a *template* for implementations of a (very simple) linked list derived type. The template is used to define specific implementations that differ in the type of data the list holds. Specific implementations are required to define a printing routine that takes care of the data in the list items.

```
! def_linkedlist.f90 --
!     Very basic implementation of linked lists
!     Main purpose: illustrate the use of templates
!
!     Real and real, dimension(:), allocatable
!
!     Comparison to pure module implementation
!
template linked_lists
    implicit none

    private
    public :: linked_list_def

    type, generic :: data_type ! No specific properties needed
    end type data_type

    type linked_list_def
        type(linked_list_def), pointer :: next
        type(data_type)                :: data
    contains
        procedure :: add   => add_to_list
        procedure :: print => print_list
    end type linked_list_def
contains
subroutine add_to_list( list, data )
    class(linked_list_def), intent(inout) :: list
    type(data_type), intent(in)           :: data

    type(linked_list_def), pointer        :: item

    allocate( item )
    item%data = data

    item%next => list%next
```

[2]The prototype program does not understand line continuations in a use_template statement – it has been introduced in the listing for readability.

```
    list%next => item
end subroutine add_to_list

subroutine print_list( list, lun )
    class(linked_list_def), intent(in), target :: list
    integer, intent(in)                        :: lun

    type(linked_list_def), pointer             :: item
    integer                                    :: i

    i = 1
    item => list
    do
        call print_item( lun, i, item%data )

        if ( associated( item%next ) ) then
            i = i + 1
            item => item%next
        else
            exit
        endif
    enddo
end subroutine print_list
end template

module linked_list_reals
    use_template linked_lists, real => data_type, &
        linked_list_real => linked_list_def
contains
subroutine print_item( lun, indx, data )
    integer, intent(in) :: lun
    integer, intent(in) :: indx
    real, intent(in) :: data

    write( lun, '(i5,a,e14.5)' ) indx, ':', data
end subroutine print_item
end module linked_list_reals

module linked_list_multireals
    use_template linked_lists, real, dimension(:), allocatable => data_type, &
        linked_list_multireal => linked_list_def
contains
subroutine print_item( lun, indx, data )
    integer, intent(in) :: lun
    integer, intent(in) :: indx
    real, dimension(:), intent(in) :: data
    write( lun, '(i5,a,5e14.5,/,(6x,5e14.5))' ) indx, ':', data
end subroutine print_item
end module linked_list_multireals

program test_linked_list
    use linked_list_reals
    use linked_list_multireals

    type(linked_list_real)             :: mylist
    type(linked_list_multireal)        :: mylist_array
    integer                            :: i, j
    real                               :: value
    real, dimension(:), allocatable    :: array

    do i = 1,10
        value = 0.1 * i
        call mylist%add( value )

        array = [(0.1 * j, j = 1,i)]
        call mylist_array%add( array )
    enddo

    open( 20, file = 'test_linked_list.out' )
    call mylist%print(20)
    call mylist_array%print(20)
end program test_linked_list
```