Subject: Accessor procedures
From: Van Snyder
References: See references section at the end

1 Number

TBD

2 Title

Accessor procedures.

3 Submitted By

J3

4 Status

For consideration.

5 Basic Functionality

Provide a form of procedure that can be invoked in what are now called variable-definition contexts. This is not a function that produces a pointer, which pointer can appear in variable-definition contexts, in the same way that "lvalue" functions are used in C[1]. It is a procedure that is in some sense the reverse of a function. When it is invoked, it receives the value to be "stored" as well as whatever arguments are specified. This is not the same as defined assignment, which is type-by-type, not object-by-object. Such procedures have appeared in a few obscure languages such as POP-2, and in the more modern (and some would argue more mainstream) C#. They are also provided in Fortran.NET by Lahey/Fujitsu (this may be due in part to the influence of the CLI undercarriage for Fortran.NET, which it shares with C#).

6 Rationale

In [5] David Parnas showed that a significant maintenance cost was caused by the fact that different data structures have different syntaxes of representation. The solution he proposed was to encapsulate reference and update operations in procedures. This results in references to the data structures (except the ones in the access procedures) having a syntax independent of their, which in turn allows to change their representations without affecting the text of the references. Although this is an advantage, for most data structures there are several disadvantages:

- One must write (at least) two procedures for each rank. These procedures usually consist mostly of the procedure header and declarations for the dummy arguments. Code bulk is the single most reliable predictor of the total "ownership" cost for software. In this case, most of the bulk is unproductive.
- Since the *execution-part*s of these procedures usually consist of little more than an assignment statement, executing them consists mostly of executing the instructions that implement the procedure call and return. That is, most of the time spent in executing them is not spent in doing what they do. Procedure call overhead is well know, and is the source for repeated requests for a standard way to recommend inlining procedures.
- The program author's intent is clear in a reference that uses a function, such as in `a = get_bank_balance ( person )`. But it is not clear in `call set_bank_balance ( person, a )`. Does the

[1]Interpretation 31 established that `lvalue` functions are not permitted.

latter accomplish `bank_balance(person) = a` or `a = bank_balance(person)` or something else? This requires those who maintain the program to keep in mind the functionality of the numerous procedures that implement the data structure, which increases maintenance cost.

Several authors ([2], [3], [4] and [6]) have (long ago) proposed a different solution to the same problem: Make references to every representation have the same syntax. Fortran is closer to this possibility than other main-stream languages because functions and arrays are both referenced with round brackets, and components and type-bound procedures are both referenced with the % symbol. One can therefore usually change an array or a structure component to a function, requiring only to change the places where it's declared. The exceptions are references to whole arrays or array sections, which will still require changes to all the references, and dummy argument declarations. This is, however, no worse than the present situation. It's not possible to change between an array and structure component, but that mistake can only be counted as water over the dam, and can't easily be repaired now.

If procedures are provided whose invocations can appear in variable-definition contexts, and they are allowed to be bound to types, one will almost be able to change an array or a structure component to an accessor or a function and its corresponding updater, without requiring any changes to any references — provided the function reference and updater reference can have the same name. The exceptions above will remain. Ironically, the exceptional case of references to array sections could be removed by implementing at least a limited form of intervals, provided Fortran's existing interval constructor — the colon — is used. See Section 25.3 of [1]. That, however, would be the topic of a separate proposal.

7 Estimated Impact

This is a modest project. Most of the changes will be in Section 12, with a few in Sections 6, 9 and 13.

8 Detailed Specification

Provide a new variety of procedure that can be invoked in a variable-definition context. One way is to provide a new variety of procedure with only the updater property, and that can be joined to a function in a generic interface. Unfortunately, generic identifiers cannot be actual arguments.

Another approach is to provide a new variety of procedure that is both a function and updater, in a single unit, with a construct that controls whether it provides a value or receives a value. Examples below assume this approach. Some obvious details are omitted; the ones presented could be changed.

No matter how it's done, it should be possible to bind the procedure to a type.

8.1 Proposed syntax

A new procedure called an *accessor* is proposed, with a procedure header similar to a *function-stmt*:

R1	*accessor-subprogram*	**is**	*accessor-stmt*
			[*specification-part*]
			provide-part
			receive-part
			[*internal-subprogram-part*]
			end-accessor-stmt
R2	*accessor-stmt*	**is**	*prefix* ACCESSOR *accessor-name* ■
			■ (*dummy-arg-name-list*) ■
			■ [TRANSFER (*transfer-name*)]

where *prefix* is the same as for a *function-stmt*. As in a *function-stmt*, if there is no explicit *transfer-name*, the *transfer-name* is the same as the *accessor-name*. Accessors are not interoperable.

R3	*provide-part*	**is**	WHEN PROVIDE
			execution-part

Control reaches this *execution-part* when the procedure is invoked in a value-providing context, such as within an expression, in an output item list, or in association with a dummy argument that does not have INTENT(OUT). The *transfer-name* behaves like a *result-name*.

R4 *receive-part* **is** WHEN RECEIVE
[*execution-part*]

Control reaches this *execution-part* when the procedure is invoked in what is now described as a variable-definition context, such as the left side of an assignment statement, an item in an input item list, or an actual argument associated with a dummy argument that does not have INTENT(IN). The value transferred into the procedure is associated with the *transfer-name*, which behaves like a dummy argument with the VALUE attribute.

References to invoke an accessor have exactly the same syntax as references to invoke a function, the only difference being that references to invoke an accessor can appear in variable-definition contexts.

8.2 Additional details

A few intrinsic functions, at least REAL (of a complex argument), AIMAG, EXPONENT, FRACTION, and maybe ABS (including perhaps for complex argument), should be changed to intrinsic accessors.

8.3 Example application

This example application (mostly stubs) provides a "persistent array," or a simulation of an "associated variable". The critical entity is the accessor `My_Var`.

```
module Associated_Variable_M

  private
  public :: Open_My_Var, Close_My_Var, My_Var, Drop_A_Few  ! procedures
  public :: BlockSize, RK                                  ! parameters
  protected :: How_Many_Blocks                             ! variables

  integer, parameter :: BlockSize = 128  ! Variables per block
  integer, parameter :: RK = kind(0.0d0) ! Kind for variables
  integer, save :: How_Many_Blocks = 0

  type :: Block_T ( K )
    integer, kind :: K
    type(block_t), pointer :: Prev, Next ! Double-linked circular list
    logical :: Dirty = .false.           ! Changed since being read from the file
    integer :: FirstOne                  ! Index of first variable in Vars
    real(k) :: Vars(BlockSize)           ! The data
  end type Block_T

  type(block_t(rk)), pointer, save :: Blocks => NULL() ! The blocks in memory

  integer, save, UnitNumber              ! of the persistent data file

  ! Put a fancy data structure here -- maybe a hash table -- to find blocks quickly.

contains

  subroutine Open_My_Var ( ... )
    ! Specify the file associated with My_Var, and open it
  end subroutine Open_My_Var ( ... )

  subroutine Close_My_Var ( ... )
    ! Flush the in-memory blocks to the file associated with My_Var and close it.
  end subroutine Close_My_Var ( ... )

```

```
  real(rk) accessor My_Var ( Index )
    integer, intent(in) :: Index
    integer :: Var_Index
    type(block_t) :: The_Block

    when provide
      call find_the_block
      my_var = the_block\%vars(var_index)
    when receive
      call find_the_block
      the_block\%vars(var_index) = my_var
      the_block\%dirty = .true.
  contains
    subroutine Find_The_Block
      ! Find the desired block, if it's in memory.
      ! If it's not in memory, read it from the opened file, if it's there.
      ! If it's not in the file, create it out of thin air and initialize
      ! the Vars field to zero.
      ! Associate The_Block with the block, and set Var_Index to the
      ! interesting subscript of the Vars field.
      ! Put the accessed block at the head of the list.
    end subroutine Find_The_Block
  end accessor My_Var

  subroutine Drop_A_Few ( N )
    integer, intent(in) :: N
    ! Write (if dirty) N (if there are that many) not-recently-used (from the
    ! end of the list) blocks to the associated file, then deallocate them.
    ! It's public so you could call it if an allocate somewhere else fails.
  end subroutine Drop_A_Few

end module Associated_Variable_M
```

9 History

10 References

1. Van Snyder, J3 paper 97-114r2, Section 25.

2. Robert M. Balzer, *Dataless programming*, in **Proceedings of the Fall Joint Computer Conference** (1967).

3. Jay Early, *Toward an understanding of data structures*, **Comm. ACM 14**, 10 (October 1971) 617-627.

4. Charles M. Geschke and James G. Mitchell, *On the problem of uniform references to data structures*, **IEEE Transactions on Software Engineering SE-2**, 1 (June 1975) 207-210.

5. David Parnas, *On the criteria for decomposing programs into modules*, **Comm. ACM 15**, 12 (December 1972) 1053-1058.

6. D. T. Ross, *Uniform referents: An essential property for a software engineering language*, **Software Engineering 1** (1969) 91-101.