— March 12, 2025 —
Proof-by-testing: Map and Accumulator Example with a peak at Fortran's type-safe templates

Magne Haveraaen[1]

Bergen Language Design Laboratory
Department of Computer Science, University of Bergen, Norway
https://bldl.ii.uib.no/

Abstract. This paper is about proof-by-testing, illustrated by two examples: a map function and an accumulator function, both on arrays. Type-safe generics plays a fundamental role in limiting the size of the dataset needed to test for correctness. It also gives a brief introduction to Fortran's proposed type-safe template mechanism.

1 Introduction

Knowing that code you use is trustworthy is central to code reuse. Here we give a lightweight introduction to the technique of *proof-by-testing* which can guarantee the correctness of black-box code by a limited set of carefully crafted tests [1,3].

The proof-by-testing technique is only assume the black box code is type-safe and deterministic. The more specific (fine-grained) the typing of each involved entity is, the more we can cut down the set of necessary tests. In the unary map example we show how to go from exhaustive testing to a single test to guarantee correctness. Such type-safety can easily be achieved using type-safe generics (type-safe templates),[1] as being proposed for the Fortran 2028 standard.[2] A similar effect of fine-grained typing can be achieved using abstract classes, but at a much higher implementation and runtime cost: the abstract classes have to be implemented whenever the code is being used for any specific purpose. Often such implementations will consist of wrapper classes and wrapper functions, just yielding runtime overheads with no additional benefit.

This introduction to proof-by-testing is centred on two examples: unary map and accumulate.

The unary map example is straight forward, and ends with a single check to prove the implemenatation correct. Unary map is equivalent to Fortran's elemental mechanism, which allows a function to be applied to each element of an array.

Accumulate, in its many instances (e.g., sum, product, max, min), is a fundamental function in numerical computing. It is a core part of matrix multiply, and central to artificial intelligence and graph algorithms. Fortran supports several accumulate functions, but only for a limited set of functions on standard arrays of intrinsic types.

Though this pamphlet uses plain array notation, the array could have been some specialised sparse data implementation, with a complicated and optimised traversal algorithm. Such data structures and algorithms are used in GraphBlas[3] [2].

The paper is organised as follows.

- In the next section we introduce type-safe templates using the proposed Fortran syntax (LFortran version).
- Section three studies proof-by-testing for a unary map example.
- Section four studies proof-by-testing for an accumulator example.
- Section 5 summarises the results.
- Appendix A sketches some problems with finite integer representations.
- Appendix B shows LFortran source code for the studied proof-by-testing examples.

[1] The C++ template mechanism does not have this property. It is only type-safe at instantiation time.
[2] See the sequence of papers 25-107 though 25-112 at https://j3-fortran.org/doc/meeting/235 for the February 2025 INCITS Fortran standards meeting.
[3] https://graphblas.org

2 Type-safe Templates

Type-safe template code is just like normal, type-safe code, except that the interpretation of some of the types and operations are *deferred* to *instantiation* time.

This allows the template implementer good control over which procedures can be used on deferred types: only the listed procedures with appropriate declarations. These procedures have to be deferred, since there is no way to a priory know what will be the instantiation type for a deferred type.

Interestingly, this allows the developer to gain fine control of the internal environment of a template. As an example, think of implementing a simple calculator for finding average speed. Instead of writing the functions using REAL, the functions can be written using specific types DISTANCE, TIME and SPEED.

```
module travel_module
    implicit none
    !! Declaration of a deferred binary function.
    !! Take arguments of type T and U and return a value of type V.
    requirement binop_r( T, U, V, op_func )
        type, deferred :: T
        type, deferred :: U
        type, deferred :: V
        pure elemental function op_func(lhs, rhs) result(res)
            type(T), intent(in) :: lhs
            type(U), intent(in) :: rhs
            type(V) :: res
        end function
    end requirement
    !! Travel template: parameterised by deferred types and functions
    template travel_tmpl( &
            DISTANCE, TIME, SPEED, plus_D, plus_T, D_divided_by_T, D_divided_by_S)
        !! Local declarations of the deferred types and functions
        require :: binop_r(DISTANCE, DISTANCE, DISTANCE, plus_D) ! Add distances
        require :: binop_r(TIME, TIME, TIME, plus_T)         ! Add time intervals
        require :: binop_r(DISTANCE, TIME, SPEED, D_divided_by_T) ! Distance divided by time gives speed
        require :: binop_r(DISTANCE, SPEED, TIME, D_divided_by_S) ! Distance divided by speed gives time
    contains
        ! Compute average speed from two legs
        ! - First leg: distance and time
        ! - Second leg: distance and time
        pure function avg_S_from_DT(d1, t1, d2, t2) result(avg)
            type(DISTANCE), intent(in) :: d1, d2
            type(TIME), intent(in) :: t1, t2
            type(SPEED) :: avg
            avg = D_divided_by_T(plus_D(d1, d2), plus_T(t1, t2))
        end function
        ! Compute average speed from two legs
        ! - First leg: distance and speed
        ! - Second leg: distance and speed
        pure function avg_S_from_DS(d1, s1, d2, s2) result(avg)
            type(DISTANCE), intent(in) :: d1, d2
            type(SPEED), intent(in) :: s1, s2
            type(SPEED) :: avg
            avg = avg_S_from_DT(d1, D_divided_by_S(d1, s1), d2, D_divided_by_S(d2, s2))
        end function
    end template
end module
!! Simple test program
program travel_test
    use travel_module
```

```
    ! Instantiating the deferred types and functions with REAL and relevant operators
    instantiate travel_tmpl( &
        real, real, real, operator(+), operator(+), operator(/), operator(/)), &
        only: avg_real_S_from_DT => avg_S_from_DT, avg_real_S_from_DS => avg_S_from_DS
    print *, "Travel test, both examples should give the same result: 0.5"
    print *, "test1=", avg_real_S_from_DT(1.0, 4.0, 3.0, 4.0)
    print *, "test2=", avg_real_S_from_DS(1.0, 0.25, 3.0, 0.75)
end program
```

As observed, this introduces more declarations than normal coding, but inside `travel_tmpl` we cannot add values of DISTANCE and TIME, nor of SPEED. We can convert DISTANCE and TIME to SPEED, as well as DISTANCE and SPEED to TIME. But we cannot confuse TIME and SPEED in the calculations due to the typing regime. This example can be tested in the LFortran web based compiler at `https://dev.lfortran.org` — and note how the compiler flags problems in the code if the code inside `travel_tmpl` is modified when trying to create expressions with improper types.

Running this example in the LFotran web compiler at `https://dev.lfortran.org` we get the following output.

```
Travel test, both examples should give the same result: 0.5
test1= 0.50000000
test2= 0.50000000

Compilation time: 12.99999999627471 ms
Execution time: 0 ms
```

Making mistakes in the source code, such as those discussed above, will give immediate feedback from the compiler. However, there are many errors that may go unnoticed. Detecting these will require more extensive tests.

3 Proof-by-testing of a unary map function

A unary map applies a function u to all elements of an array, producing a new array with modified values. This is similar to Fortran applying an elemental function to all elements of an array. The Fortran mechanism is limited to standard arrays, while a user defined unary map will be needed for user defined sparse array implementations [2].

Here we explain proof-by-testing of a unary map irrespective of the data structure chosen to represent the array, and independently of the traversal algorithm. A discussion of proof-by-testing of this and other array algorithms can b found in [3].

Consider implementing a function $c : I[] \to I[]$ that takes an array of integers I as input and returns another array where these elements have changed sign—an application of unary minus to every element in the input array. We can write a template version of this by deferring the choice of function (change sign) and the types of the array elements involved. In fact, we may make the choice of deferred types and unary function even more fine-grained. Define a templated function $c\langle T, U, u\rangle : T[] \to U[]$ where the two types T, U and unary function $u : T \to U$ are deferred. The function c should, for every index $i : I$ of the input array $t : T[]$ yield an output value $c(t)[i] = u(t[i])$, i.e., a generalised version of the previously described "change sign function".

Given a type-safe black-box implementation of c, we know the following.

- The function c cannot produce an output element of type U without calling u on an input array element.
  There is no way for the algorithm of c to magically invent a type U value since it has no clue what U is. The only accessible way to create a value of type U is by calling u, which has to be applied to a value of type T exactly once. The only accessible values of type T are values in the input array.
- The algorithm of c cannot depend on the values of the elements of the input array.
  The algorithm can (and will have to) depend on the structure, i.e., the shape or size, of the

input array, but it has no way of gauging what a T (nor U) value is, besides that it can apply u to a T value in order to produce a U value.

Compare this knowledge to what the algorithm could do if it was written for arrays of integers. As output elements it would be able to provide arbitrary constants, do any computation on any combination of input shape and element values.

The listed knowledge about the type-safe black-box algorithm c gives us leeway in crafting a dataset for testing correctness. The only ways teh templated c can fail are the following cases.

- It places an output element in a wrong position compared to the input element.
- It skips some input elements.
- It uses the same input element multiple times, possibly in non-consecutive positions in the output array.
- The output array may end up with a wrong number of elements due to skipping/repetition of output values.
- Such mistakes may be specific for each array shape.

For any given array shape, we can construct the follow test which will prove correctness of c. Instantiate both T and U to the integers I, and use the identity function $\mathrm{id} : I \to I$, defined by $\mathrm{id}(i : I) = i$, as u. Fill the test array $t : T[]$ with distinct values, e.g., $t[i] = i$. Then the check $c\langle I, I, \mathrm{id}\rangle(t) = t$ will ensure the correctness of c for the given array size.

If we want to be certain that c will function correctly in an application, we will need to reapply the test above for each array shape used by the application. This is obviously a finite number, and for any typical application it will be less than a handfull of distinct shapes. The application may not know the actual array shapes in advance, as these may be dependent on the input data. In such a case, the application can run this check as an integrated safety procedure for each caase as they arise.

The above argument builds on the parametricity of the type-safe implementation [1]. This says that every instantiation of the code uses the same algorithm, i.e., the algorithm will not adapt to the knowledge of the actual arguments for the deferred types and operations.[4]

4 Proof-by-testing of a black box implementation of accumulate

The accumulate function folds the values in an array down to a single value. It can do so by applying a binary function to the elements of the array. If we apply an operation which is associative and commutative, the order we apply it to the array elements does not matter. Mathematical functions, such as addition and multiplication are associative and commutative, hence sum and product are well defined independently of which order addition and multiplication are applied to the array elements. However, if we wanted to apply an operation like subtraction, we would have to be really careful in defining which order the subtractions should take place on the array elements.

In the remainder of this note, we will assume the binary function for accumulate will be associative and commutative. Then we will not have to worry about which order the operation is performed. We can still instantiate the template with a function that is neither associative nor commutative, such as subtraction, but the result of accumulate would then be very dependent on its undisclosed algorithm.

Instead of asking for specific implementations of the accumulate function, let us write a *contract* for a type-safe generic (black-box) implementation of accumulate, $a\langle T, b\rangle : T[] \to T$. Here T is the deferred type of the data, and $b : T, T \to T$ is an associative, commutative binary function that should be used for accumulating the values of the input array elements. We want a to compute something like $b(t[1], b(t[2], b(t[3], \ldots)))$ for $t : T[]$. Since the contract says that b is associative and commutative, it does not matter which order the computation takes. Computing $b(b(t[3], b(t[2], t[1])), \ldots)$ will yield the same answer as the specification.

[4] This principle is clearly broken by the C++ template mechanism which allows *template specialisation* based on these arguments.

This algorithm can then be instantiated for computing the sum or product of some number type, or a user defined associative and commutative function on a user defined type.

Since we are working with the same type T both for the input array and for the output value, the type system gives us far less control of the internals of the accumulate function than it did for the unary map function c above.

- A faulty implemented a may skip some array elements in the application of f.
- A faulty a may repeatedly use an element $t[i]$ when computing the result.
  Specifcially for a one-element input array $t : T[1]$ the result should be $t[1]$, but a may miscompute and deliver, e.g., $f(f(t[1], t[1]), t[1])$.
- The function a is allowed to traverse the data in an arbitrary order, but this order can only be determined by the shape of the array (its size).
  As for the unary map above, the algorithm of a has no way of gauging the array values in determining how to compute.

4.1 Proof-by-testing of a mathematical algorithm a

Now if a was a mathematical function, we could instantiate it with T as the natural numbers N and b as multiplication. Then $a\langle N, *\rangle$ should compute the product of the elements of the argument array. We can easily check this, by filling the array with distinct prime numbers. The result should be a number that factors uniquely into the distinct prime numbers of our test array. If a factor is ommited or occurs more than once, we can definetely deduce that a does not fulfill its contract. And since a cannot be sensitive to the fact that it is asked to produce the product of certain primes, a correct answer is *proof* that the algorthim is correct—for the provided array size. Hence, we have a proof-by-testing of a's correctness for this shape array.

Note that we do not know in which order the product was computed. This is OK, since we explicitly in the contract for a have agreed that b is associative and commutative.

4.2 Proof-by-testing of a computer algorithm a

The proof strategy above starts from the natural numbers being a *unique factorisation domain*[5] (UFD): every natural number is the unique product of a multiset of prime numbers.

On a computer we do not have the mathematical numbers to play with. Instead we have approximations like the modulus integers or saturation integers, see appendices A.1 and A.2, respectively. Modulo integers do not have the UFD property. For saturation integers and overflow trapping integers, we can assume the UFD property since we can detect if an out-of-bounds computation has occured. However, this only works for small arrays with far less than 30 elements, much too small for interesting datasets.

A possible approach is to create a suite of test arrays with integers modulus 4, type M_4. There should be one test array t_i for each index i. Each such test array should contain only the number 1, except for the element at index i which should contain the number 2. The product of every such array should be $a\langle M_4, *\rangle(t_i) \cong_4 2$, but if the implementation a has an error, the result will be 1 or 0.

- The product $a\langle M_k, *\rangle(t_i) \cong_4 1$ if the algorithm a skips element i.
- The product $a\langle M_k, *\rangle(t_i) \cong_4 0$ if the element i is included multiple times in the computation.

Using the typical computer integers of modulus 2^k for any positive k such as $32, 64, 128$, we can let the computer do its normal modulus computation and then do modulus 4. In these cases, we will have the same observations as discribed in the list above.

The test data set with such a t_i for every index i will uncover any problems with the algorithm a. Recall that the algorithm cannot depend on the contents of any test array. It is only dependent on the shape of the array. And this will hold for any array data structure layout and any accumulation

[5] `https://en.wikipedia.org/wiki/Unique_factorization_domain`

algorithm, as long as the implementation is type-safe and fully parameterised by the array element type and a binary function. Unfortunately, we may need a rather large collection of test arrays for large array sizes.

5 Summary

Proving software correct is a longstanding and difficult problem. Formal verification requires a precise semantics of the programming language coupled with their encoding for verification tools.

In this paper we have studied crafting test datasets for some basic array algorithms: unary map and accumulate. Running the resulting test sets on an algorithm will prove its correctness. The assumption is that the algorithm is type-safe and parameterised by the types and functions it uses on the array alements. Such parameterisation is known as type-safe templates or type-safe generics. Otherwise the data structure layout and algorithms can be totally opaque for proof-by-testing.

References

1. Bernardy, J.P., Jansson, P., Claessen, K.: Testing polymorphic properties. In: Gordon, A. (ed.) Programming Languages and Systems: Proceedings of the 19th European Symposium on Programming (ESOP 2010). Lecture Notes in Computer Science, vol. 6012, pp. 125–144. Springer-Verlag (2010), `https://doi.org/10.1007/978-3-642-11957-6_8`
2. Davis, T.A.: Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra. ACM Trans. Math. Softw. 45(4), 44:1–44:25 (2019), `https://doi.org/10.1145/3322125`
3. Haveraaen, M.: Proving a core code for FDM correct by $2 + dw$ tests. In: Scholz, S., Shivers, O. (eds.) Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY@PLDI 2018, Philadelphia, PA, USA, June 19, 2018. pp. 42–49. ACM (2018), `https://doi.org/10.1145/3219753.3219759`

A Finite integer computer representations

A.1 Modulo integers and unique factorisation

Here we take a brief look at why modulus integers do not have unique factorisations.

Let us take a look at modulo $16 = 2^4$ integers (4bit integers). The multiplication table is given below.

	0	1	**2**	**3**	4	**5**	6	**7**	8	9	A	**B**	C	**D**	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	0	2	4	6	8	A	C	E
3	0	3	6	9	C	F	2	5	8	B	E	1	4	7	A	D
4	0	4	8	C	0	4	8	C	0	4	8	C	0	4	8	C
5	0	5	A	F	4	9	E	3	8	D	2	7	C	1	6	B
6	0	6	C	2	8	E	4	A	0	6	C	2	8	E	4	A
7	0	7	E	5	C	3	A	1	8	F	6	D	4	B	2	9
8	0	8	0	8	0	8	0	8	0	8	0	8	0	8	0	8
9	0	9	2	B	4	D	6	F	8	1	A	3	C	5	E	7
A	0	A	4	E	8	2	C	6	0	A	4	E	8	2	C	6
B	0	B	6	1	C	7	2	D	8	3	E	9	4	F	A	5
C	0	C	8	4	0	C	8	4	0	C	8	4	0	C	8	4
D	0	D	A	7	4	1	E	B	8	5	2	F	C	9	6	3
E	0	E	C	A	8	6	4	2	0	E	C	A	8	6	4	2
F	0	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1

Here we observe that multiplying with even numbers gives a repetitive pattern, while multiplication with odd numbers gives a permutation of the numbers. Studying the table, we note that $1 \cong_{16} 3*B \cong_{16} 5*D \cong_{16} 7^2$. Iterating these expressions an arbitrary number of times will not change the result of a product. Further, $1 \cong_{16} 9^2$, and since $9 \cong_{16} 3^2 \cong_{16} 5^2 \cong_{16} B^2 \cong_{16} D^2 \cong_{16} 3*5*7$, any product of two such factors will yield 1 and therefore is not noticable if part of a miscalculation. Now the accumulate algorithm does not know which number is placed where in the array, so chances are we would detect such an error in the algorithm by rearranging the data. However, our analysis of how to create a sufficient set of test data arrays to ensure correctness becomes significantly more complex.

Looking into numbers modulus $4294967296 = 2^{32}$ (32bit integers), we get similar patterns. The analysis of where problems may lie becomes significantly more difficult. The error situations may seem implausible, as for instance for the number 17 we find that $1 \cong_{4294967296} 17^{268435456}$. Though this may seem like an outragous miscalculation, for an array with 10GB of 32bit integers, the number of computational steps here is of the same magnitude as doing an actual multiplication of all array elements. Thus the extra computational effort needed for the error may not stand that clearly out from the intended computation.

Enlarging the range for modulus numbers will thus not escape the problem of undiscovered errors in the computation. Whether the probability of undiscovered errors remaning is acceptably low requires a deeper study of approximate prime factorisation properties for arbitrarily sized modulus numbers. Especially interesting, for current computers, are establishing such properties for modulus 2^{32}, 2^{64} and 2^{128}.

A.2 Alternative integer approximations and unique factorisation

As alternatives to the prevalent modulus semantics for our computers' integer approximations, we may consider saturation integers or error detection on overflow.

Saturation integers have range bounds, and the computation saturates (stays at the maximum value) when the bound is reached. Using 4bit integers the saturation value is at F. Thus multiplications with three will be $3*4 \cong_{s16} C$, $3*5 \cong_{s16} F$ and $3*6 \cong_{s16} F$.

Another alternative is to trap the computation on overflow. Then we will get an error message for overflow rather than compute the wraparound $1 \cong_{16} 3*B$.

This does not help much for proof by testing. For 32bit, modulus numbers $2^{32} = 4294967296$, we find that multiplying the first nine primes, $2, 3, 5, 7, 11, 13, 17, 19, 23$ gives 223092870, but multiplying this with the tenth prime, 29, yields 6469693230, going beyond the representable 32bit numbers. Enlarging the range to 64bit, modulus numbers $2^{64} = 18446744073709551616$, we saturate when multiplying the first 16 primes, the product of the 15 first being 614889782588491410. For 128bit integers we saturate when multiplying the first 27 primes.

Thus using saturation integers or trap on overflow, we will detect any problems with the accumulate algorithm by looking at the factorisation of the results. Unfortunately, this only works for small arrays.

B Fortran code for the verification examples

B.1 Module with Reusabe Requirements

Requirements for binary and unary functions. For the unary functions, there are versions with 1D array arguments.

```
module Requirements_m

    !! Declaration of a unary function.
    !! Take argument of type T and return a value of type U.
    requirement unop_r( T, U, op_func )
        type, deferred :: T
        type, deferred :: U
```

```
        pure elemental function op_func(lhs) result(res)
            type(T), intent(in) :: lhs
            type(U) :: res
        end function
    end requirement

    !! Declaration of a unary function on arrays.
    !! Take argument of type T(:) and return a value of type U.
    requirement unopa_r( T, U, op_func )
        type, deferred :: T
        type, deferred :: U
        pure function op_func(lhs) result(res)
            type(T), intent(in) :: lhs(:)
            type(U) :: res
        end function
    end requirement

    !! Declaration of a unary function on arrays with array return.
    !! Take argument of type T(:) and return a value of type U(:).
    requirement unopaa_r( T, U, op_func )
        type, deferred :: T
        type, deferred :: U
        pure function op_func(lhs) result(res)
            type(T), intent(in) :: lhs(:)
            type(U) :: res
        end function
    end requirement

    !! Declaration of a binary function.
    !! Take arguments of type T and U and return a value of type V.
    requirement binop_r( T, U, V, op_func )
        type, deferred :: T
        type, deferred :: U
        type, deferred :: V
        pure elemental function op_func(lhs, rhs) result(res)
            type(T), intent(in) :: lhs
            type(U), intent(in) :: rhs
            type(V) :: res
        end function
    end requirement

    !! Copy (and possibly) convert data.
    !! Take argument of type U and output data to argument of type T.
    requirement copy_r( T, U, copy )
        type, deferred :: T
        type, deferred :: U
          pure elemental subroutine copy(lhs, rhs)
            type(T), intent(out) :: lhs
            type(U), intent(in) :: rhs
          end subroutine
    end requirement
end module
```

B.2 Accumulator and Unary Map

The template code for the *accumulate*[6] and *unary map* functions.

[6] The unary map has an extra type parameter which will be explained in the pamplat on *accumulate accuracy*.

```
module ArraySupport_m
    use Requirements_m
    implicit none
    private
    public :: accumulator_t, unary_map_t

    template accumulator_t ( U, T, copy, fadd )
        require :: binop_r ( U, T, U, fadd)
        require :: copy_r ( U, T, copy )
      contains
        function accumulate( arg ) result (res)
            type(T), intent(in) :: arg(:)
            type(U) :: res

            type(U) :: acc
            integer :: i
            ! print *, ".. accumulate"
            call copy ( acc, arg(1) )
            do i = 2, size(arg)
              acc = fadd ( acc, arg(i) )
            end do
            res = acc
        end function
    end template

    template unary_map_t ( T, U, f )
        require :: unop_r ( T, U, f )
      contains
        function umap( arg ) result (res)
            type(T), intent(in) :: arg(:)
            type(U) :: res( size(arg) )

            type(U) :: tmp( size(arg) )
            integer :: i
            ! print *, ".. umap"
            do i = 1, size(arg)
              tmp(i) = f ( arg(i) )
            end do
            res = tmp
        end function
    end template

end module
```

B.3 Test Harnesses for Verification

Here we have some crafted test harnesses for the accumulate and unary map functions. Both have the actual accumulate and unary map functions as template parameters, while the involved types and data sets are defined internally.

There are also some checks that appropriate instantiations of these functions are being used, i.e., accumulate on multiplication (product) and unary map on the identity function.

```
module VerificationHarness_m
  use Requirements_m
  integer, parameter :: validarray = [7,1,1,2,1,3,5,1,1]

    template test_harness_accumulator_t ( accumulate )
        require :: unopa_r ( integer, integer, accumulate)
```

```
  contains
    function check_accumulate( arg ) result(res)
        type(integer), intent(in) :: arg(:)
        type(logical) :: res

        type(integer) :: checkarray(size(arg))
        integer :: i
        logical :: allok
        ! print *, ".. check_accumulate for ", size(arg)
        ! print *, ".. validation: ", accumulate([7,1,1,2,1,3,5,1,1])
        if ( 210 /= accumulate([7,1,1,2,1,3,5,1,1]) ) then
          stop "VERIFICATION ERROR test_harness_accumulator_t: " &
          // "Accumulate function is not a product! " &
          // "Instantiate it with operator(*)"
        end if
        allok = .TRUE.
        do i = 1, size(arg)
          checkarray = 1
          checkarray(i) = 2
          allok = allok .and. accumulate( checkarray ) == 2
          ! print *, allok, " i=", i, " check=", accumulate( checkarray )
          ! print *, checkarray
        end do
        res = allok
    end function
  end template


  template test_harness_unarymap_t ( umap )
    require :: unop_r ( integer, integer, umap )
  contains
    function check_umap ( arg ) result(res)
        type(integer), intent(in) :: arg(:)
        type(logical) :: res

        type(integer) :: checkarray(size(arg))
        integer :: i
        ! print *, ".. check_umap for ", size(arg)
        if ( validarray /= umap ( validarray ) ) then
          stop "VERIFICATION ERROR test_harness_unarymap_t: " &
          // "Umap function is not an identity map! " &
          // "Instantiate it with function id_integer"
        end if
        do i = 1, size(arg)
          checkarray(i) = i
        end do
        res = all ( checkarray == umap ( checkarray ) )
    end function
  end template


contains

  pure elemental function id_integer ( arg ) result(res)
    type(integer), intent(in) :: arg
    type(integer) :: res
    res = arg
  end function
```

```
   pure elemental function double_integer ( arg ) result(res)
       type(integer), intent(in) :: arg
       type(integer) :: res
       res = 2 * arg
   end function

end module
```

At the bottom of the module some support functions have been added: the identity function and a doubling function on integers.

B.4 Test program

This is the test program for the accumulate and unary map functions.

It instantiates the proper test version of the accumulate function using multiplication on the integers, and the correspniong test verification harness.

Due to some unresolved problems with the handling of deferred functions with array arguments in LFortran, the unary map test is not set up as intended, but an ad hoc variant is intriduced.

```
program ArraySupportTest

  use ArraySupport_m
  use copy_m
  use VerificationHarness_m

  integer :: data15(15)
  integer :: i
  do i = 1, size(data15)
    data15(i) = i
  end do
  integer :: data20000(20000)

  print *, "ArraySupportTest Program"

  instantiate accumulator_t ( integer, integer, copy_ii, operator(*) ), only : my_prod_ii => accumulate
  instantiate test_harness_accumulator_t ( my_prod_ii ), only : check_accumulate
  print *, "Verifying the accumulate function"
  print *, "small check (accumualte) ", check_accumulate (data15)
  print *, "large check (accumulate) ", check_accumulate (data20000)

  instantiate unary_map_t ( integer, integer, id_integer ), only : id_integer_umap => umap
  ! instantiate test_harness_unarymap_t ( id_integer_umap ), only : id_check_umap => check_umap
  instantiate test_harness_unarymap_t ( id_integer ), only : id_check_umap => check_umap
  print *, "Fake: Verifying the umap function"
  print *, "small check (umap)      ", id_check_umap (data15)
  print *, "large check (umap)      ", id_check_umap (data20000)
  print *, "small check (umap crafted) ", all ( data15 == id_integer_umap(data15) )

  instantiate unary_map_t ( integer, integer, double_integer ), only : double_integer_umap => umap
  ! instantiate test_harness_unarymap_t ( integer, integer, double_integer_umap ), only : double_integer_umap =
  instantiate test_harness_unarymap_t ( double_integer ), only : double_check_umap => check_umap
  print *, "Demonstrating internal fail safe in verification test harness"
  print *, "small print (umap identity) ", id_integer_umap (data15)
  print *, "small print (umap double) ", double_integer_umap (data15)
  print *, "small print (elem double) ", double_integer (data15)
  print *, "small check (umap double) ", double_check_umap (data15)

  print *, "Verfification completed"
```

```
contains

end program
```

The test program also has a failed test where the instantiation of unary map uses the doubling function rather than the intended identity function. This activiates the fail safe mechanism in the test harness, explaining what the proper instantiation of the unary map function should be.

Below is the output from running the test program.

```
ArraySupportTest Program
Verifying the accumulate function
small check (accumualte)  True
large check (accumulate)  True
Fake: Verifying the umap function
small check (umap)        True
large check (umap)        True
small check (umap crafted) True
Demonstrating internal fail safe in verification test harness
small print (umap identity) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
small print (umap double) 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
small print (elem double) 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
STOP VERIFICATION ERROR test_harness_unarymap_t: Umap function is not an identity map!
    Instantiate it with function id_integer
```