VHDL-200x Data Types and Abstractions White Paper 1

Type Genericity

Peter Ashenden, Ashenden Designs
peter@ashenden.com.au

Version 1, 28-Sep-03

Abstract

This white paper proposes a design for type-genericity extensions to VHDL. The extensions are based on a design developed as part of the SUAVE project by Peter Ashenden while at the University of Adelaide and Phil Wilsey at the University of Cincinnati. That design, in turn, was strongly influenced by the type-genericity features of the Ada programming language.

Revision History

Version 1. 28-Sep-03, Peter Ashenden. Initial version based on SUAVE Language Description.

1 Introduction

Reuse of a design unit can be improved by making it applicable in a wider set of contexts, for example, by making it more generic. VHDL currently includes a mechanism, generic constants, that allows components and entities to be parameterized with formal constants. Actual generic constants are specified when components are instantiated and when entities are bound. The generic constant mechanism is widely used to specify timing parameters and array port bounds, among other things.

In this proposal we extend the generic mechanism of VHDL to improve support for reuse. There are two main aspects to the extension. The first is to allow subprograms and packages to have generic interface clauses. The second is to allow formal types in a generic interface clause, making the generic item reusable for a variety of different types. Formal subprograms and formal packages are also allowed as a corollary to allowing formal types.

A further extension that would be required if process declaration were included in the language would be the inclusion of formal processes in generic clauses. This would parallel formal subprograms, allowing specification of action processes for instances of generic units.

In this white paper, we present the syntax and static semantics of generic units. In BNF syntax rules, we underline those rules or parts of rules that are extensions to existing VHDL rules. We start by describing the extended forms of package and subprogram declarations that include formal generic lists. We also describe the way in which generic packages and subprograms may be instantiated with a generic map aspect providing actuals for the formal generics. We then present a detailed description of the various forms of formal generic declaration, illustrating them with examples.

Note that this white paper focuses on proposed extensions to the generics mechanism in relative isolation. The SUAVE proposal makes further extensions. Some of those extensions are intended to make generic units more usable. An example is inclusions of package declarations and package instances in any declarative part, not just as design units. Other extensions arise from interactions with additional language features proposed in SUAVE. An example is formal generics supporting derived types. We do not present any of those extensions in this white paper, preferring to focus just on what is essential for extended generics.

2 Generic Packages

We extend the declaration of packages to allow inclusion of a formal generic clause. The extended syntax rule for a package declaration is shown is:

package_declaration ::=
package identifier **is**
[*formal*_generic_clause]
package_declarative_part
end [**package**] [*package*_simple_name] ;

A package that includes a formal generic clause is a generic package. A generic package is a template for an ordinary package, and does not provide declarations itself. It must be instantiated as described below. Examples of packages with formal generic clauses are shown in subsequent sections.

2.1 Instantiating a Generic Package

In order to make use of a generic package, it must be instantiated to associate actuals with the formal generics. The syntax rule is:

generic_package_instantiation ::=
package identifier **is new** *generic_package*_name
[generic_map_aspect] ;

An instance of a generic package is semantically equivalent to a normal package as currently defined in VHDL. Examples of generic package instantiation are shown in subsequent sections. A generic package is instantiated as a design unit. The extended syntax rule is:

primary_unit ::=
. . .
| generic_package_instantiation

3 Generic Subprograms

We also extend the declaration of subprograms to allow inclusion of a formal generic clause. The extended syntax rule for a subprogram specification is:

subprogram specification ::=
procedure designator
[**generic** (generic_list)]
[[**parameter**] (formal_parameter_list)]
| [**pure** | **impure**] **function** designator
[**generic** (generic_list)]
[[**parameter**] [(formal_parameter_list)] **return** type_mark

The extended syntax allows for an option **parameter** keyword before the parameter list. This is to make the declaration syntactically consistent with other units that include generic clauses.

A subprogram declaration or body that includes a formal generic clause in its specification is a generic subprogram. A generic subprogram is a template for an ordinary subprogram. It cannot be called, but must be instantiated as described in elow. Examples of subprograms with formal generic clauses are shown in subsequent sections.

If a generic subprogram is declared as a separate subprogram declaration and subprogram body, the subprogram body must include the formal generic clause, which must conform with the formal generic clause in the subprogram declaration.

3.1 Instantiating a Generic Subprogram

A generic subprogram may be instantiated as a subprogram. The syntax rule is:

generic_subprogram_instantiation ::=
subprogram_kind designator **is new** *generic_subprogram*_name
[generic_map_aspect] ;

Generic subprograms may be instantiated in any declarative part in which subprograms may be declared. Examples of generic subprogram instantiation are shown in subsequent sections. The extended syntax rules are:

entity_declarative_item ::=
. . .
| generic_subprogram_instantiation

block_declarative_item ::=
. . .
| generic_subprogram_instantiation

process_declarative_item ::=
. . .
| generic_subprogram_instantiation

subprogram_declarative_item ::=
. . .
| generic_subprogram_instantiation

package_declarative_item ::=
. . .
| generic_subprogram_instantiation

package_body_declarative_item ::=
. . .
| generic_subprogram_instantiation

4 Extended Generic Maps

A generic map aspect is used to associate actual generics with formal generics upon instantiation of a generic subprogram, a generic package, a process, a component or an entity. Agenericmap is also used to associate actual generics with formal generics in a block statement.

The extended syntax rule for an actual designator, allowing specification of actual generics for formal type, subprogram and package generics, is:

actual_designator ::=
. . .
| type_mark
| *subprogram*_name
| *package_instance*_name

For a formal generic type, the associated actual type is designated by a type mark. For a formal generic subprogram, the associated actual subprogram is designated by a subprogram name. For a formal generic package, the associated actual package is designated by the name of a package instance. Examples of extended generic maps are shown in subsequent sections.

5 Extended Generic Clauses

We extend the kinds of formal generics that may be specified in a formal generic clause to include formal types, formal subprograms and formal packages. These can be included in generic clauses of package declarations, subprogram specifications, block statements, entity declarations and component declarations. The revised syntax rule is:

interface_declaration ::=
. . .
| interface_type_declaration
| interface_subprogram_declaration
| interface_package_declaration

Interface type, subprogram and package declarations may only appear in formal generic clauses. A formal generic clause may only include interface constant, type, subprogram and package declarations.

The rule (LRM ¶4.3.2.1) that prohibits use of an item declared in an interface list within the declaration of other items in the interface list is relaxed in the case of generic interface lists. Items declared in a generic interface list may be used in the declaration of items declared subsequently in the interface list.

A further change is a relaxation of the rule (LRM ¶??.??) that prohibits interface constants from being of access types. We allow constant parameters of subprograms and impure functions to be of access types, provided ... (contitions to be determined...).

6 Formal Generic Types

An interface type declaration defines a formal generic type that can be used to pass a particular type when the generic unit is instantiated. The form of the interface type definition determines the "shape" of the type, that is, the class of type that can be passed as the actual generic type. There is a shape corresponding to each of the classes of type provided by VHDL. The syntax rules are:

interface_type_declaration ::=
    type identifier **is** interface_type_definition

interface_type_definition ::=
    interface_private_type_definition
    | interface_discrete_type_definition
    | interface_integer_type_definition
    | interface_physical_type_definition
    | interface_floating_type_definition
    | interface_array_type_definition
    | interface_record_type_definition
    | interface_access_type_definition
    | interface_file_type_definition
    | interface_protected_type_definition

6.1 Formal Private Types

An interface private type definition defines a formal generic type that can denote any type. The generic unit can only assume that operations available for all types are applicable, namely, variable assignment, equality and inequality operations. The syntax rule is:

interface_private_type_definition ::=
    private

Example

A package defining an ADT for sets of elements can be made reusable by making it generic with respect to element type, as shown below. The formal type generic element_type represents the element type, and the actual associated with it can be any type.

```
package sets is
    generic ( type element_type is private );
    type set;
    -- element_node and structure of set are private
    type element_node is record
        next_element : set;
        value : element_type;
    end record element_node;
    type set is access element_node;
    constant empty_set : set;
    procedure copy ( from : in set; to : out set );
    function "+"( R : element_type ) return set; -- singleton set
    impure function "+"( L : set; R : element_type ) return set; -- add to set
    impure function "+"( L : element_type; R : set ) return set; -- add to set
    impure function "+"( L, R : set ) return set; -- set union
    . . .
end package sets;
package body sets is
    constant empty_set :set := null;
    . . .
end package body sets;
```

Given a type thingy, an ADT for sets of elements of this type may instantiated as follows:

```
package thingy_sets is new work.sets
    generic map ( element_type => thingy );
```

———

6.2 Formal Discrete Types

An interface discrete type definition defines a formal generic type that denotes any discrete type. The generic unit can assume that operations available for any discrete type are applicable. For example, the unit can use the 'succ and 'pred attributes, and can use the formal type as an index type for an array. The syntax rule is:

interface_discrete_type_definition ::= (<>)

Example

The following entity declaration describes a counter that counts through successive values of any discrete type denoted by count_type:

```
entity counter is
    generic ( type count_type is (<>) );
    port ( clk : in bit; data : out count_type );
end entity counter;
```

An architecture body for the counter is shown below. Since count_type denotes a discrete type, the process can use the attributes 'low, 'high and 'succ.

```
architecture behavioral of counter is
begin
    count_behavior : process is
        variable count : count_type := count_type'low;
    begin
        data <= count;
        wait until clk = '1';
        if count = count_type'high then
            count := count_type'succ(count);
        else
            count := count_type'low;
        end if;
    end process count_behavior;
end architecture behavioral;
```

Some examples of instantiation of this counter are:

```
type state_type is ( idle, receiving, processing, replying );
. . .
natural_counter : entity work.counter(behavioral)
    generic map ( count_type => natural )
    port map ( clk => master_clk, data => natural_data );
state_counter : entity work.counter(behavioral)
    generic map ( count_type => state_type)
    port map ( clk => master_clk, data => state_data );
```

———

6.3 Formal Integer Types

An interface integer type definition defines a formal generic type that denotes any integer type. The generic unit can assume that operations available for any integer type are applicable. For example, the unit can use the predefined integer arithmetic operators on values of the type. The syntax rule is:

interface_integer_type_definition ::= **range** <>

Example

The counter example can be rewritten to use a formal integer type, allowing use of the addition operator in the implementation.

```
entity counter is
    generic ( type count_type is range <> );
    port ( clk : in bit; data : out count_type );
end entity counter;

architecture behavioral of counter is
begin
    count_behavior : process is
        variable count : count_type := count_type'low;
    begin
        data <= count;
        wait until clk = '1';
        if count = count_type'high then
            count := count + 1;
        else
            count := count_type'low;
        end if;
    end process count_behavior;
end architecture behavioral;
```

———

6.4 Formal Physical Types

An interface physical type definition defines a formal generic type that denotes any physical type. The generic unit can assume that operations available for any physical type are applicable. For example, the unit can use the predefined arithmetic operators that operate on values of physical types. The syntax rule is:

interface_physical_type_definition ::= **units <>**

Example

The following generic package defines a physical type that is the dimensional product of two physical types specified as formal physical type generics:

```
package product_measures is
    generic ( type measure1, measure2 is units <> );
    type product_measure is units
        product_unit;
        product_unit_E3 = 1E3 product_unit;
        product_unit_E6 = 1E6 product_unit;
        product_unit_E9 = 1E9 product_unit;
        product_unit_E12 = 1E12 product_unit;
    end units product_measure;
    function "*"( L : measure1; R : measure2 ) return product_measure;
    function "/"( L : product_measure; R : measure1 ) return measure2;
    function "/"( L : product_measure; R : measure2 ) return measure1;
end package product_measures;
```

The implementation of the operators is completed in the package body:

```
package body product_measures is
    function "*"( L : measure1; R : measure2 ) return product_measure is
    begin
        return product_measure'val( measure1'pos(L) * measure2'pos(R) );
    end function "*";
    function "/"( L : product_measure; R : measure1 ) return measure2 is
    begin
```

```
        return measure2'val( product_measure'pos(L) / measure1'pos(R) );
    end function "/";
    function "/"( L : product_measure; R : measure2 ) return measure1 is . . .
end package body product_measures;
```

An instantiation of the package to define a type for power as the product of voltage and current is shown below.

```
package voltage_pkg is
    type voltage is units
        uV;
        mV = 1000 uV;
        V = 1000 mV;
    end units voltage;
end package voltage_pkg;
package current_pkg is
    type current is units
        uA;
        mA = 1000 uA;
        A = 1000 mA;
    end units current;
end package current_pkg;
use work.voltage_pkg.all, work.current_pkg.all;
package power_measures is new work.product_measure
    generic map ( measure1 => voltage, measure2 => current );
package power_aliases is
    alias power is work.power_measures.product_measure;
    alias pW is work.power_measures.product_unit;
    alias nW is work.power_measures.product_unit_E3;
    alias uW is work.power_measures.product_unit_E6;
    alias mW is work.power_measures.product_unit_E9;
    alias W is work.power_measures.product_unit_E12;
end package power_aliases;
```

Use of the "*" and "/" functions defined by the package instance performs dimensionally correct arithmetic on voltage, current and power values. For example:

```
use work.voltage_pkg.all, work.current_pkg.all,
    work.power_measures.all, work.power_aliases.all;
variable heater_voltage : voltage;
variable heater_current : current;
. . .
if heater_voltage * heater_current > 100 mW then . . .
```

———

6.5 Formal Floating Types

An interface floating type definition defines a formal generic type that denotes any floating-point type. The generic unit can assume that operations available for any floating-point type are applicable. For example, the unit can use the predefined floating-point arithmetic operators on values of the type. The syntax rule is:

interface_floating_type_definition ::= **range** <> . <>

Since use of use of a formal floating-point type generic typically also involves use of a formal package generic, we defer an example to Section ??.

6.6 Formal Array Types

An interface array type definition defines a formal generic type that denotes any array type. The generic unit can assume that operations available for the array type are applicable. For example, the unit can perform array indexing, slicing and concatenation operations on values of the type, and can refer to array attributes. The syntax rule is:

interface_array_type_definition ::= array_type_definition

A formal array type and the associated actual array type must both be constrained or both be unconstrained. Both must have the same dimensionality, the same index types in each dimension and the same element types. For a formal constrained array type, the index constraint must be specified in the form of a type mark, and the actual array type must have the same index range as the formal array type.

In order to satisfy the above rules, the index type and element types are typically declared as formal generic types earlier in the generic list. Then, when the generic unit is instantiated, the index type and element type of the actual array type are provided in the generic map as well as the actual array type.

Example

The following entity declaration describes a shift register that stores and shifts a vector of arbitrary type:

```
entity shift_register is
    generic ( type index_type is (<>);
              type element_type is private;
              type vector is array ( index_type range <> ) of element_type );
    port ( clk : in bit;
           data_in : in element_type;
           data_out : out vector );
end entity shift_register
```

The architecture body is:

```
architecture behavioral of shift_register is
begin
    shift_behavior : process is
        constant data_low : index_type := data_out'low;
        constant data_high : index_type := data_out'high;
        type ascending_vector is array ( data_low to data_high ) of element_type;
        variable stored_data : ascending_vector;
    begin
        data_out <= stored_data;
        wait until clk = '1';
        stored_data(data_low to index_type'pred(data_high))
            := stored_data(index_type'succ(data_low) to data_high);
        stored_data(data_high) := data_in;
    end process shift_behavior;
end architecture behavioral;
```

The entity can be instantiated as follows:

```
signal master_clk, carry_in : bit;
signal result : bit_vector(15 downto 8);
. . .
bit_vector_shifter : entity work.shift_register(behavioral)
    generic map ( index_type => natural, element_type => bit, vector => bit_vector )
    port map ( clk => master_clk, data_in => carry_in, data_out => result );
```

———

6.7 Formal Record Types

To be completed...

6.8 Formal Access Types

An interface access type definition defines a formal generic type that denotes any access type.

The syntax rule is:

interface_access_type_definition ::= access_type_definition

Example

The following generic procedure copies the value of one dynamic vector to another. The index type and element type of the dynamic vectors are specified as formal generic types, and the dynamic vector type is represented as a pointer to an allocated array.

```
procedure copy_vector
    generic ( type index_type is (<>);  type element_type is private;
              type vector is array ( index_type range <> ) of element_type;
              type vector_ptr is access vector )
    parameter ( src : in vector_ptr; dest : inout vector_ptr ) is
begin
    if dest /= null then
        deallocate ( dest );
    end if;
    dest := new vector'(src.all);
end procedure copy_vector;
```

Given the following declarations for dynamic vectors of time values:

```
type time_vector is array ( natural range <> ) of time;
type time_vector_ptr is access time_vector;
variable schedule1 : time_vector_ptr := new time_vector'(1 ns, 3 ns, 10 ns);
variable schedule2 : time_vector_ptr;
```

The procedure may be instantiated and called as follows:

```
procedure copy_time_vector is new copy_vector
    generic map ( index_type => natural, element_type => time,
                  vector => time_vector, vector_ptr => time_vector_ptr );
. . .
copy_time_vector ( src => schedule1, dest => schedule2 );
```

———

6.9 Formal File Types

An interface file type definition defines a formal generic type that denotes any file type. The syntax rule is:

interface_file_type_definition ::= file_type_definition

Example

VHDL does not allow a file to contain elements that are multidimensional arrays. One means of working around this restriction is to use a file of the element type of the multidimensional array, and to read and write array elements in sequence. The following package provides read and write operations using this approach for two-dimensional arrays. The package is generic with respect to the array type, and includes a file type with the same element type as the array type. The package cannot be written with a private type for the element type, since there are restrictions on the kinds of types that can be included as file elements. This example uses a floating-point type as the element type. Similar packages could be written for other kinds of types that are permissible for file elements.

```
package floating_matrix_IO is
    generic ( type row_index_type is (<>);  type col_index_type is (<>);
              type element_type is (<>.<>);
              type matrix is array ( row_index_type, col_index_type ) of element_type;
              type matrix_file is file of element_type );
    procedure read ( file f : matrix_file; value : out matrix );
```

```
    procedure write ( file f : matrix_file; value : in matrix );
end package floating_matrix_IO;
```

An implementation of the read and write operations is shown in the following package body.

```
package body floating_matrix_IO is
    procedure read ( file f : matrix_file; value : out matrix ) is
    begin
        for row_index in row_index_type loop
            for col_index in col_index_type loop
                read ( f, value(row, col) );
            end loop;
        end loop;
    end procedure read;
    procedure write ( file f : matrix_file; value : in matrix ) is
    begin
        for row_index in row_index_type loop
            for col_index in col_index_type loop
                write ( f, value(row, col) );
            end loop;
        end loop;
    end procedure write;
end package body floating_matrix_IO;
```

An example of instantiation of this package is:

```
package transformation_pkg is
    subtype transformation_index is integer range 1 to 3;
    type transformation_matrix is array ( transformation_index, transformation_index ) of real;
    type real_file is file of real;
end package transformation_pkg;
use work.transformation_pkg.all;
package transformation_matrix_IO is new work.floating_matrix_IO
    generic map ( row_index_type => transformation_index,
                  col_index_type => transformation_index,
                  element_type => real,
                  matrix => transformation_matrix,
                  matrix_file => real_file );
```

The instance can then be used as follows:

```
use work.transformation_pkg.all, transformation_matrix_IO.all;
file transformation_file : real_file;
variable next_transformation : transformation_matrix;
. . .
file_open ( transformation_file, “test_transformations.dat”, read_mode );
read ( transformation_file, next_transformation );
```

————

6.10 Formal Protected Types

To be completed...

7 Formal Generic Subprograms

An interface subprogram declaration defines a formal generic subprogram that can be used to pass a particular subprogram when the generic unit is instantiated. The syntax rule is:

interface_subprogram_declaration ::=
    subprogram_specification [**is** subprogram_default]

subprogram_default ::= name | **<>**

The subprogram specification may not contain a generic clause. The subprogram default specifies the subprogram to use if no actual generic subprogram is provided on instantiation. If a name is specified as the subprogram default, it must denote a callable subprogram with the same signature as that of the subprogram specification. If a box (**<>**) is specified as the subprogram default, it indicates that the actual generic subprogram should be a subprogram that is directly visible at the point of instantiation and that has the same name and signature as those of the subprogram specification.

Example

The following package defines an ADT for lookup tables. A table contains elements that are each identified by a key value. The formal function key_of determines the key for a given element. No default function is provided, so the user must supply an actual function on instantiation of the package. The formal function "<" is used to compare key values. The default function is specified using the "**<>**"notation, so if an appropriate function named "<" is visible at the point of instantiation, no actual need be specified. The generic procedure traverse is parameterized by an action procedure. An instance of traverse applies the actual action procedure to each element in the table.

```
package lookup_tables is
    generic ( type element_type is private;
                type key_type is private;
                function key_of ( E : element_type ) return key_type;
                function "<"( L, R : key_type ) return boolean is <> );
    type lookup_table;
    -- tree_record and structure of lookup_table are private
    type tree_record is record
        left_subtree, right_subtree : lookup_table;
        element : element_type;
    end record tree_record;
    type lookup_table is access tree_record;
    procedure lookup ( table : in lookup_table; lookup_key : in key_type;
                        element : out element_type; found : out boolean );
    procedure search_and_insert ( table : in lookup_table; element : in element_type;
                                  already_present : out boolean );
    procedure traverse
        generic ( procedure action ( element : in element_type ) )
        parameter ( table : in lookup_table );
end package lookup_tables;
```

The package body is shown below. The formal functions **key_of** and "<" are invoked using the formal name.

```
package body lookup_tables is
    procedure lookup ( table : in lookup_table; lookup_key : in key_type;
                        element : out element_type; found : out boolean ) is
        variable current_subtree : lookup_table := table;
    begin
        found := false;
        while current_subtree /= null loop
            if lookup_key < key_of( current_subtree.element ) then
                lookup ( current_subtree.left_subtree, lookup_key, element, found );
            elsif key_of( current_subtree.element ) < lookup_key then
                lookup ( current_subtree.right_subtree, lookup_key, element, found );
            else
                found := true;
                element := current_subtree.element;
                return;
            end if;
```

```
        end loop;
    end procedure lookup;
    procedure search_and_insert ( table : in lookup_table; element : in element_type;
                                  already_present : out boolean ) is ...
    procedure traverse
        generic ( procedure action ( element : in element_type ) )
        parameter ( table : in lookup_table ) is
    begin
        if table = null then
            return;
        end if;
        traverse ( table.left_subtree );
        action ( table.element );
        traverse ( table.right_subtree );
    end procedure traverse;
end package body lookup_tables;
```

Suppose a model requires a lookup table of test patterns that use character strings as keys. Such a table may be instantiated as shown below. Since the predefined function "<"operating on strings is visible at the point of instantiation, it is used as the actual function for the formal function "<".

```
package test_pattern_pkg is
    type test_pattern_type is . . .
    function test_id_of ( test_pattern : in test_pattern_type ) return string;
end package test_pattern_pkg;
use work.test_pattern_pkg.all;
package test_pattern_tables is new work.lookup_tables
    generic map ( element_type => test_pattern_type,
                  key_type => string,
                  key_of => test_id_of );
```

The traversal procedure can be used to count the number of elements in the table by instantiating it as follows:

```
use work.test_pattern_pkg.all, test_pattern_tables.all;
variable count : natural := 0;
procedure count_a_test_pattern ( test_pattern : in test_pattern_type ) is
begin
    count := count + 1;
end procedure count_a_test_pattern;
procedure count_test_patterns is new traverse
    generic map ( action => count_a_test_pattern );
```

The instantiated traversal function can be called with a test pattern lookup table as a parameter, as follows:

```
variable patterns_to_apply : lookup_table;
. . .
count_test_patterns ( patterns_to_apply );
```

———

8 Formal Generic Packages

An interface package declaration defines a formal generic package that can be used to pass a particular instance of a generic package when the generic unit is instantiated. Formal generic packages are typically used where a generic package needs to make use of declarations from another generic package. In such cases, an instance of the second generic package is provided as an actual for a formal generic of the first package. The syntax rule is:

interface_package_declaration ::=
    package identifier **is**
        new *generic_package*_name interface_package_actual_part ;

interface_package_actual_part ::=
    generic map (<>)
    | [generic_map_aspect]

The name must denote a generic package. If the interface package actual part is of the form that includes a box (<>), the actual package may be any instance of the named generic package. If the interface package actual part is a generic map aspect, the actual package must be an instance of the named generic package with the same actual generics as those specified in the generic map aspect. If the interface package actual part is empty, the actual package must be an instance of the named generic package with the same actual generics as the defaults for the generic package.

The following example is adapted from the Ada Rationale [??].

Example

Suppose a generic package for complex numbers is defined as follows:

```
package generic_complex_numbers is
    generic ( type float_type is range <>.<> );
    type complex is record
        re, im : float_type;
    end record complex;
    function "+"( L, R : complex ) return complex;
    function "–"( L, R : complex ) return complex;
. . .
end package generic_complex_numbers;
```

The package is generic so that it may be used with different floating point types. A package for generic complex vectors can be defined as shown below. It is generic with respect to the type of complex number used as vector elements. The package could be defined with the complex type as a formal type generic, but then the operators needed to implement the vector functions would also have to be included as formal subprogram generics. A more succinct form is to specify a formal package generic for the package defining the complex number ADT.

```
package generic_complex_vectors is
    generic ( package complex_numbers is new work.generic_complex_numbers
                    generic map (<>) );
    use complex_numbers.all;
    type complex_vector is array ( natural range <> ) of complex;
    function "+"( L, R : complex_vector ) return complex_vector;
    function "–"( L, R : complex_vector ) return complex_vector;
    . . .
end package generic_complex_vectors;
```

As an illustration of how the operations defined in the formal complex number ADT package are used, the package body for the complex vectors package is as follows:

```
package body generic_complex_vectors is
    function "+"( L, R : complex_vector ) return complex_vector is
        alias L_norm : complex_vector(1 to L'length) is L;
        alias R_norm : complex_vector(1 to R'length) is R;
        variable result : complex_vector(1 to L'length);
    begin
        assert L'length = R'length
            report "Addition of complex vectors of different lengths";
        for index in result'range loop
            result(index) := L_norm(index) + R_norm(index);
```

```
        end loop;
        return result;
    end function "+";
    function "–"( L, R : complex_vector ) return complex_vector is . . .
    . . .
end package body generic_complex_vectors;
```

Suppose now that the complex numbers package is instantiated as follows:

```
package float_pkg is
    type short_float is range –10.0 to 10.0;
end package float_pkg;
package short_complex_numbers is new work.generic_complex_numbers
    generic map ( float_type => work.float_pkg.short_float );
```

The complex vectors package is then instantiated as follows:

```
package short_complex_vectors is new work.generic_complex_vectors
    generic map ( complex_numbers => work.short_complex_numbers );
```

Now suppose further that a generic package for mathematical functions on floating point types is defined as follows:

```
package generic_float_functions is
    generic ( type float_type is range <>.<> );
    function sqrt ( x : float_type ) return float_type;
    function log ( x : float_type ) return float_type;
    . . .
end package generic_float_functions;
```

A generic package for mathematical functions on complex numbers can be defined as shown below. The formal package generic complex_numbers is an instance of the generic complex_numbers package defined above. That package is generic with respect to the underlying floating point type used. The complex functions package must use the generic floating point functions package, but instantiated with the same underlying floating point type. This is enforced by the actual part specified for the formal package generic_float_functions.

```
package generic_complex_functions is
    generic ( package complex_numbers is new work.generic_complex_numbers
                generic map (<>);
              package float_functions is new work.generic_float_functions
                generic map ( float_type => complex_numbers.float_type ) );
    use complex_numbers.all;
    function sqrt ( x : complex ) return complex;
    function log ( x : complex ) return complex;
    . . .
end package generic_complex_functions;
```

This package can be instantiated for the complex number type as follows:

```
package short_float_functions is new work.generic_float_functions
    generic map ( float_type => work.float_pkg.short_float );
package short_complex_functions is new generic_complex_functions
    generic map ( complex_numbers => work.short_complex_numbers,
                  float_functions => work.short_float_functions );
```

———