INTERNATIONAL STANDARD

ISO/IEC 13813

First edition
1998-08-01

Information technology — Programming languages — Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types)

Technologies de l'information — Langages de programmation — Paquetages génériques de déclarations de types réel et complexe et opérations de base pour Ada (y compris les types vecteur et matrice)

Reference number
ISO/IEC 13813:1998(E)

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 13813 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee 22, *Programming languages, their environments and system software interfaces*.

Annexes A, B, C, D and E form an integral part of this International Standard. Annexes F, G and H are for information only.

Introduction

The generic packages described here are intended to provide the basic real and complex scalar, vector, and matrix operations from which portable, reusable applications can be built. This International Standard serves a broad class of applications with reasonable ease of use, while demanding implementations that are of high quality, capable of validation and also practical given the state of the art.

The specifications included in this International Standard are presented as compilable Ada specifications in annexes A, B, C, D and E with explanatory text in numbered sections in the main body of text. The explanatory text is normative, with the exception of notes (labeled as such).

The word "may," as used in this International Standard, consistently means "is allowed to" (or "are allowed to"). It is used only to express permission, as in the commonly occurring phrase "an implementation may"; other words (such as "can," "could" or "might") are used to express ability, possibility, capacity or consequentiality.

Information technology — Programming languages — Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types)

1 Scope

This International Standard defines the specifications of three generic packages of scalar, vector and matrix operations called `GENERIC_COMPLEX_TYPES`, `GENERIC_REAL_ARRAYS` and `GENERIC_COMPLEX_ARRAYS`, the specification of a package of related exceptions called `ARRAY_EXCEPTIONS` and the specification of a generic package of complex input and output operations called `COMPLEX_IO`. A package body is not required for `ARRAY_EXCEPTIONS`; bodies of the other packages are not provided by this International Standard.

The specifications of non-generic packages called `COMPLEX_TYPES`, `REAL_ARRAYS` and `COMPLEX_ARRAYS` are also defined, together with those of analogous packages for other precisions. This International Standard does not provide the bodies of these packages.

This International Standard specifies certain fundamental scalar, vector and matrix arithmetic operations for real, imaginary and complex numbers. They were chosen because of their utility in various application areas; moreover, they are needed to support a generic package for complex elementary functions.

This International Standard is applicable to programming environments conforming to ISO/IEC 8652.

NOTE — This International Standard is specifically designed for applicability in programming environments conforming to ISO/IEC 8652:1987. Except for the packages and generic packages dealing with arrays, comparable facilities are specified in ISO/IEC 8652:1995; specifications for the generic array packages conforming to ISO/IEC 8652:1995 are provided in annex G.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 8652, *Information technology — Programming languages — Ada.*

ISO/IEC 11430, *Information technology — Programming languages — Generic package of elementary functions for Ada.*

ISO/IEC 11729, *Information technology — Programming languages — Generic package of primitive functions for Ada.*

ISO/IEC 13814, *Information technology — Programming languages — Generic package of complex elementary functions for Ada.*

3 Types and operations provided

The following record type, scalar type and four array types are exported by the packages provided by this International Standard:

COMPLEX	IMAGINARY
REAL_VECTOR	REAL_MATRIX
COMPLEX_VECTOR	COMPLEX_MATRIX

Type COMPLEX provides a cartesian representation of complex scalars; type IMAGINARY is provided to represent pure imaginary scalars; two composite types with elements of type REAL are provided, REAL_VECTOR and REAL_MATRIX, to represent real vectors and matrices; and two composite types with elements of type COMPLEX are provided, COMPLEX_VECTOR and COMPLEX_MATRIX, to represent complex vectors and matrices.

The following twenty-four operations are provided:

"+"	"-"	"*"	"/"
"<"	"<="	">"	">="
"**"	"abs"	CONJUGATE	TRANSPOSE
RE	IM	SET_RE	SET_IM
COMPOSE_FROM_CARTESIAN	MODULUS	ARGUMENT	COMPOSE_FROM_POLAR
UNIT_VECTOR	IDENTITY_MATRIX	GET	PUT

These are the usual mathematical operators (+, -, * and /) for real, complex and imaginary scalars, and for real and complex vectors and matrices (together with analogous componentwise operations for vectors); the relational operators (<, <=, > and >=) for imaginary scalars; the exponentiation operator (**) for complex and imaginary scalars, and for real and complex vectors; the absolute value operator (abs) for real, imaginary and complex scalars, and for real and complex vectors and matrices; the conjugate operation (CONJUGATE) for complex and imaginary scalars, and for complex vectors and matrices; the transpose operation (TRANSPOSE) for real and complex matrices; the cartesian component-part operations (RE, IM, SET_RE, SET_IM and COMPOSE_FROM_CARTESIAN) for complex scalars, vectors and matrices (and, where applicable, for imaginary scalars), for selecting component-parts and for composing from component-parts; the polar component-part operations (MODULUS, ARGUMENT and COMPOSE_FROM_POLAR) for complex scalars, vectors and matrices, for selecting component-parts and for composing from component-parts; the initializing operations (UNIT_VECTOR and IDENTITY_MATRIX) for real and complex vectors and matrices; and the input/output operations (GET and PUT) for complex scalars.

4 Instantiations

This International Standard describes generic packages GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS, GENERIC_COMPLEX_ARRAYS and COMPLEX_IO. Each package has a generic formal parameter, which is a generic formal floating-point type named REAL. At instantiation, this parameter determines the precision of the arithmetic.

This International Standard also describes non-generic packages COMPLEX_TYPES, REAL_ARRAYS and COMPLEX_ARRAYS, which provide the same capability as instantiations of the packages GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS and GENERIC_COMPLEX_ARRAYS. It is required that non-generic packages be constructed for each precision of floating-point type defined in package STANDARD.

Depending on the implementation, the user may or may not be allowed to specify a generic actual type having a range constraint (see clause 5). If allowed, such a range constraint will have the usual effect of causing CONSTRAINT_ERROR to be raised when a scalar argument outside the user's range is passed in a call to one of the subprograms, or when one of the subprograms attempts to return a scalar value (or to construct a composite value with a scalar component or element) outside the user's range. Allowing the generic actual type to have a range constraint also has some implications for implementers.

5 Implementations

Portable implementations of GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS, GENERIC_COMPLEX_ARRAYS and COMPLEX_IO are strongly encouraged. However, implementations are not required to be portable. In particular, an implementation of this International Standard in Ada may use pragma INTERFACE or other pragmas, unchecked conversion, machine-code insertions, or other machine-dependent techniques as desired.

An implementation is allowed to make reasonable assumptions about the environment in which it is to be used, but only when necessary in order to match algorithms to hardware characteristics in an economical manner. For example, an implementation is allowed to limit the precision it supports (by stating an assumed maximum value for SYSTEM.MAX_DIGITS), since portable implementations would not, in general, be possible otherwise. All such limits and assumptions shall be clearly documented. By convention, an implementation of GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS, GENERIC_COMPLEX_ARRAYS or COMPLEX_IO is said not to conform to this International Standard in any environment in which its limits or assumptions are not satisfied, and this International Standard does not define its behavior in that environment. In effect, this convention delimits the portability of implementations.

For any of the generic packages GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS, GENERIC_COMPLEX_ARRAYS or COMPLEX_IO, an implementation may impose a restriction that the generic actual type shall not have a range constraint that reduces the range of allowable values. If it does impose this restriction, then the restriction shall be documented, and the effects of violating the restriction shall be one of the following:

Compilation of a unit containing an instantiation of that generic package is rejected.

CONSTRAINT_ERROR or PROGRAM_ERROR is raised during the elaboration of an instantiation of that generic package.

Conversely, if an implementation does not impose the restriction, then such a range constraint shall not be allowed, when included with the user's actual type, to interfere with the internal computations of the subprograms; that is, if the arguments and result (of functions), or their components, are within the range of the type, then the implementation shall return the result (if any) and shall not raise an exception (such as CONSTRAINT_ERROR).

Any of the restrictions discussed above may in fact be inherited from implementations of the package GENERIC_ELEMENTARY_FUNCTIONS of ISO/IEC 11430 and the package GENERIC_PRIMITIVE_FUNCTIONS of ISO/IEC 11729, if used. The dependence of an implementation on such inherited restrictions should be documented.

Implementations of GENERIC_COMPLEX_TYPES, GENERIC_REAL_ARRAYS and GENERIC_COMPLEX_ARRAYS shall function properly in a tasking environment. Apart from the obvious restriction that an implementation of these packages shall avoid declaring variables that are global to the subprograms, no special constraints are imposed on implementations. With the exception of COMPLEX_IO, nothing in this International Standard requires the use of such global variables.

Some hardware and their accompanying Ada implementations have the capability of representing and discriminating between positively and negatively signed zeros as a means (for example) of preserving the sign of an infinitesimal quantity that has underflowed to zero. Implementations of these packages may exploit that capability, when available, so as to exhibit continuity in the results of ARGUMENT as certain limits are approached. At the same time, implementations in which that capability is unavailable are also allowed. Because a definition of what comprises the capability of representing and distinguishing signed zeros is beyond the scope of this International Standard, implementations are allowed the freedom not to exploit the capability, even when it is available. This International Standard does not specify the signs that an implementation exploiting signed zeros shall give to zero results; it does, however, specify that an implementation exploiting signed zeros shall yield a scalar result (or a scalar element of a composite result) for ARGUMENT that depends on the sign of a zero imaginary component of a scalar argument (or a corresponding scalar element of a composite argument). An implementation shall exercise its choice consistently, either exploiting signed-zero behavior everywhere or nowhere in these packages. In addition, an implementation shall document its behavior with respect to signed zeros.

In implementations of GENERIC_COMPLEX_TYPES and GENERIC_COMPLEX_ARRAYS, all operations involving mixed real and complex arithmetic are required to construct the result by using real arithmetic (instead of by converting real values to complex values and then using complex arithmetic). This is to facilitate conformance with IEEE arithmetic.

6 Exceptions

The ARGUMENT_ERROR exception is declared in GENERIC_COMPLEX_TYPES and GENERIC_COMPLEX_ARRAYS. This exception is raised by a subprogram in these generic packages when the argument(s) of the subprogram violate one or more of the conditions given in the subprogram's definition (see clause 8).

NOTE — These conditions are related only to the mathematical definition of the subprogram and are therefore implementation independent.

The ARRAY_INDEX_ERROR exception is declared in GENERIC_REAL_ARRAYS and GENERIC_COMPLEX_ARRAYS. This exception is raised by a subprogram in these generic packages when the argument(s) of the subprogram violate one or more of the conditions for matching elements of arrays (as in predefined equality); that is, for dyadic array operations, the bounds of the given left and right array operands need not be equal, but their appropriate vector lengths or row and/or column lengths (for matrices) shall be equal.

The ARGUMENT_ERROR and ARRAY_INDEX_ERROR exceptions are declared as renamings of exceptions of the same name declared in the ELEMENTARY_FUNCTIONS_EXCEPTIONS package of ISO/IEC 11430 and in the ARRAY_EXCEPTIONS package of this International Standard, respectively. These exceptions distinguish neither between different kinds of argument errors or array index errors, nor between different subprograms. The ARGUMENT_ERROR exception does not distinguish between instantiations of either GENERIC_COMPLEX_TYPES, GENERIC_COMPLEX_ARRAYS, the GENERIC_ELEMENTARY_FUNCTIONS package of ISO/IEC 11430 or the GENERIC_COMPLEX_ELEMENTARY_FUNCTIONS package of ISO/IEC 13814. The ARRAY_INDEX_ERROR exception does not distinguish between different instantiations of either GENERIC_REAL_ARRAYS or GENERIC_COMPLEX_ARRAYS.

Besides ARGUMENT_ERROR and ARRAY_INDEX_ERROR, the only exceptions allowed during a call to a subprogram in these packages are predefined exceptions, as follows:

— Virtually any predefined exception is possible during the evaluation of an argument of a subprogram in these packages. For example, NUMERIC_ERROR, CONSTRAINT_ERROR, or even PROGRAM_ERROR could be raised if an argument has an undefined value; and, as stated in clause 4, if the implementation allows range constraints in the generic actual type, then CONSTRAINT_ERROR will be raised when the value of an argument lies outside the range of the user's generic actual type. Additionally, STORAGE_ERROR could be raised, e.g. if insufficient storage is available to perform the call. All these exceptions are raised before the body of the subprogram is entered and therefore have no bearing on implementations of these packages.

— For the subprograms in COMPLEX_IO only, any of the exceptions declared (by renaming) in TEXT_IO may be raised in the appropriate circumstances. For example, TEXT_IO.LAYOUT_ERROR will be raised during an output operation to a string if the given string is too short to hold the formatted output. Additionally, TEXT_IO.DATA_ERROR will be raised during the evaluation of arguments of an input operation if the components of the complex value obtained are not of the type REAL, or, for implementations of COMPLEX_IO not based on an instantiation of TEXT_IO.FLOAT_IO, if the input sequence does not have the required syntax. Implementations of COMPLEX_IO which make use of an instantiation of TEXT_IO.FLOAT_IO shall make every attempt to raise TEXT_IO.DATA_ERROR in the presence of invalid input sequence syntax; however, this International Standard recognizes the difficulty in handling all possible invalid input sequences for these types of implementations.

— Also, as stated in clause 4, if the implementation allows range constraints in the generic actual type, then CONSTRAINT_ERROR will be raised when a subprogram in these packages attempts to return a scalar value (or to construct a composite value with a scalar component or element) outside the range of the user's generic actual type. The exception raised for this reason shall be propagated to the caller of the subprogram.

— Whenever the arguments of a subprogram are such that a scalar result (or a scalar component or element of a composite result) permitted by the accuracy requirements would exceed REAL'SAFE_LARGE in absolute value, as formalized below in clause 10, an implementation may raise (and shall then propagate to the caller) the exception specified by Ada for signaling overflow.

— Once execution of the body of a subprogram has begun, an implementation may propagate STORAGE_ERROR to the caller of the subprogram, but only to signal the unexpected exhaustion of storage. Similarly, once execution of the body of a subprogram has begun, an implementation may propagate PROGRAM_ERROR to the caller of the subprogram, but only to signal errors made by the user of these packages.

No exception is allowed during a call to a subprogram in these packages except those permitted by the foregoing rules. In particular, for arguments for which all scalar results (or scalar components or elements of all composite results) satisfying the accuracy requirements remain less than or equal to `REAL'SAFE_LARGE` in absolute value, a subprogram shall locally handle an overflow occurring during the computation of an intermediate result, if such an overflow is possible, and not propagate an exception signaling that overflow to the caller of the subprogram.

The only exceptions allowed during an instantiation of `GENERIC_COMPLEX_TYPES`, `GENERIC_REAL_ARRAYS`, `GENERIC_COMPLEX_ARRAYS` or `COMPLEX_IO`, including the execution of the optional sequence of statements in the body of the instance, are `CONSTRAINT_ERROR`, `PROGRAM_ERROR` and `STORAGE_ERROR`, and then only for the following reasons. The raising of `CONSTRAINT_ERROR` during instantiation is only allowed when the implementation imposes the restriction that the generic actual type shall not have a range constraint, and the user violates that restriction (it may, in fact, be an inescapable consequence of the violation). The raising of `PROGRAM_ERROR` during instantiation is only allowed for the purpose of signaling errors made by the user – for example, violation of this same restriction, or of other limitations of the implementation. The raising of `STORAGE_ERROR` during instantiation is only allowed for the purpose of signaling the exhaustion of storage.

NOTE — In ISO/IEC 8652:1987, the exception specified for signaling overflow or division by zero is `NUMERIC_ERROR`, but ISO/IEC 8652:1995 replaces that by `CONSTRAINT_ERROR`.

7 Arguments outside the range of safe numbers

ISO/IEC 8652 fails to define the result safe interval of any basic or predefined operation of a real subtype when the absolute value of one of its operands exceeds the largest safe number of the operand subtype. (The failure to define a result in this case occurs because no safe interval is defined for the operand in question.) In order to avoid imposing requirements that would, consequently, be more stringent than those of Ada itself, this International Standard likewise does not define the result of a contained subprogram when the absolute value of one of its scalar arguments (or one of the scalar components or elements of composite arguments) exceeds `REAL'SAFE_LARGE`. All of the accuracy requirements and other provisions of the following clauses are understood to be implicitly qualified by the assumption that scalar subprogram arguments (or scalar components or elements of composite subprogram arguments) are less than or equal to `REAL'SAFE_LARGE` in absolute value.

8 Method of specification of subprograms

Some of the subprograms have two or more overloaded forms. For each form of a subprogram covered by this International Standard, the subprogram is specified by its parameter and result type profile, the domain of its argument(s) if restricted, its range if restricted, and the accuracy required of its implementation. The meaning of, and conventions applicable to, the domain, range and accuracy specifications are described below.

The specification of each subprogram covered by this International Standard includes, where necessary, a characterization of the argument values for which the subprogram is mathematically defined. It is expressed by inequalities or other conditions which the arguments shall satisfy to be valid. Whenever the arguments fail to satisfy all the conditions, the implementation shall raise `ARGUMENT_ERROR`. It shall not raise that exception if all the conditions are satisfied.

Inability to deliver a result for valid arguments because the scalar result (or a scalar component or element of the composite result) overflows, for example, shall not raise `ARGUMENT_ERROR`, but shall be treated in the same way that Ada defines for its predefined floating-point operations (see clause 10).

The usual mathematical meaning of the "range" of a function is the set of values into which the function maps the values in its domain. Some of the subprograms covered by this International Standard (for example, `ARGUMENT`) are mathematically multivalued, in the sense that a given argument value can be mapped by the subprogram into many different result values. By means of range restrictions, this International Standard imposes a uniqueness requirement on the results of multivalued functions, thereby reducing them to single-valued functions.

The range of each subprogram result is shown, where necessary, in the specifications. Range definitions take the form of inequalities limiting the results of a subprogram. An implementation shall not exceed a limit of the range when that limit is a safe number of REAL (like 0.0, 1.0, or CYCLE/2.0 for certain values of CYCLE). On the other hand, when a range limit is not a safe number of REAL (like π, or CYCLE/2.0 for certain other values of CYCLE), an implementation may exceed the range limit, but may not exceed the safe number of REAL next beyond the range limit in the direction away from the interior of the range; this is, in general, the best that can be expected from a portable implementation. Effectively, therefore, range definitions have the added effect of imposing accuracy requirements on implementations above and beyond those presented as error bounds in the specifications (see clause 9).

9 Accuracy requirements

Because they are implemented on digital computers with only finite precision, the subprograms provided in these generic packages can, at best, only approximate the corresponding mathematically defined operations.

The accuracy requirements contained in this International Standard define the latitude that implementations are allowed in approximating the intended precise mathematical result with floating-point computations. Accuracy requirements of two kinds are stated in the specifications. Additionally, range definitions impose requirements that constrain the values implementations may yield, so the range definitions are another source of accuracy requirements (in that context, the precise meaning of a range limit that is not a safe number of REAL, as an accuracy requirement, is discussed in clause 8). Every result returned by a subprogram is subject to all of the subprogram's applicable accuracy requirements, except in the one case described in clause 12. In that case, the scalar result (or scalar components or elements of the composite result) will satisfy a small absolute error requirement in lieu of the other accuracy requirements defined for the subprogram.

The accuracy requirements on array operations are defined in terms of corresponding accuracy requirements on their (real or complex) scalar elements, unless the mathematical definition of the operation includes an inner product (indicated in the specifications as such). The accuracy of operations involving inner products is beyond the scope of this International Standard, except that an implementation shall document what, if any, extended-precision accumulation of intermediate results is used to implement such inner products.

The first kind of (scalar) accuracy requirement used in the specifications is a "maximum relative error requirement." It is specified by bounds on appropriate measures of the relative error in the computed result of a subprogram, which shall hold (except as provided by the rules in clauses 10 and 12) for all arguments satisfying the conditions in the domain definition, whenever those measures are defined.

Three forms of measure are used in the specifications; they depend on the type (real, imaginary or complex) of the scalar result. In the real or imaginary case, the measure is the usual "relative error"; in the complex case, the measure used for each component-part is, whenever possible, a "component-part error," but in cases where substantial cancellation may be involved this is relaxed to a "box error."

For a real result, if the mathematical result is α and the computed result is x, then the relative error $rel\_err(x)$ is defined in the usual way:

$$rel\_err(x) = |\alpha - x|/|\alpha|$$

provided the mathematical result is finite and nonzero.

For a complex result, if the mathematical result is $\alpha + i\beta$ and the computed result is $x + iy$, then the component-part errors $real\_comp\_err(x)$, $imag\_comp\_err(y)$ are defined as:

$$real\_comp\_err(x) = |\alpha - x|/|\alpha|$$

provided the mathematical component-part α is finite and nonzero, and

$$imag\_comp\_err(y) = |\beta - y|/|\beta|$$

provided the mathematical component-part β is finite and nonzero; and the box errors $real\_box\_err(x)$, $imag\_box\_err(y)$ are defined as:

$$real\_box\_err(x) = |\alpha - x|/\max(|\alpha|, |\beta|)$$

$$imag\_box\_err(y) = |\beta - y|/\max(|\alpha|, |\beta|)$$

provided the mathematical component-parts α, β are finite and not both zero.

In all other cases, the above measures of the relative error are not defined (i.e., when the mathematical result, or a component-part of the mathematical result, is infinite or zero).

The second kind of (scalar) accuracy requirement used in the specifications is a stipulation, usually in the form of an equality, that the implementation shall deliver "prescribed results" for certain special arguments. It is used for two purposes:

- to define the computed result when one of the measures of the relative error is undefined, i.e., when the mathematical result (or a component-part of the mathematical result) is zero; and

- to strengthen the accuracy requirements at special argument values.

When such a prescribed result (or component-part of a prescribed result) is a safe number of `REAL` (like `0.0`, `1.0` or `CYCLE/2.0` for certain values of `CYCLE`), an implementation shall deliver that value. On the other hand, when a prescribed result (or component-part of a prescribed result) is not a safe number of `REAL` (like π, or `CYCLE/2.0` for certain other values of `CYCLE`), an implementation may deliver any value in the surrounding safe interval. Prescribed results take precedence over maximum relative error requirements but never contravene them. Complex results need not have the same kind of accuracy requirement for both of their component-parts. Where all results of an operation are prescribed, the operation is specified as "exact."

Range definitions in the specifications are an additional source of accuracy requirements, as stated in clause 8. As an accuracy requirement, a range definition has the effect of eliminating some of the values permitted by the maximum relative error requirements, e.g. those outside the range.

10 Overflow

Floating-point hardware is typically incapable of representing numbers whose absolute value exceeds some implementation-defined maximum. For the type `REAL`, that maximum will be at least `REAL'SAFE_LARGE`. For the subprograms defined by this International Standard, whenever the maximum relative error requirements permit a scalar result (or a scalar component or element of a composite result) whose absolute value is greater than `REAL'SAFE_LARGE`, the implementation may

- yield any result permitted by the maximum relative error requirements, or

- raise the exception specified by Ada for signaling overflow.

In addition, some of the functions are allowed to signal overflow for certain arguments for which neither component of the result can overflow. This freedom is granted for operations involving either an inner product or complex exponentiation. Permission to signal overflow in these cases recognizes the difficulty of avoiding overflow in the computation of intermediate results, given the current state of the art.

NOTES

1 The rule permits an implementation to raise an exception, instead of delivering a result, for arguments for which the mathematical result (or a component-part of the mathematical result) is close to but does not exceed `REAL'SAFE_LARGE` in absolute value. Such arguments must necessarily be very close to an argument for which the mathematical result (or a component-part of the mathematical result) does exceed `REAL'SAFE_LARGE` in absolute value. In general, this is the best that can be expected from a portable implementation with a reasonable amount of effort.

2 The rule is motivated by the behavior prescribed by ISO/IEC 8652 for the predefined operations. That is, when the set of possible results of a predefined operation includes a number whose absolute value exceeds the implementation-defined maximum, the implementation is allowed to raise the exception specified for signaling overflow instead of delivering a result.

3 In ISO/IEC 8652:1987, the exception specified for signaling overflow is `NUMERIC_ERROR`, but ISO/IEC 8652:1995 replaces that by `CONSTRAINT_ERROR`.

11 Infinities

An implementation shall raise the exception specified by Ada for signaling division by zero in the following specific cases where the corresponding mathematical results, or component-parts thereof, are infinite:

a) division by (real, imaginary or complex) zero;

b) array operations whose mathematical definition involves division of an element by (real or complex) zero;

c) exponentiation of (real, imaginary or complex) zero by a negative (integer) exponent;

d) array operations whose mathematical definition involves exponentiation of (real or complex) zero by a negative (integer) exponent;

NOTE — In ISO/IEC 8652:1987, the exception specified for signaling division by zero is `NUMERIC_ERROR`, but ISO/IEC 8652:1995 replaces that by `CONSTRAINT_ERROR`.

12 Underflow

Floating-point hardware is typically incapable of representing nonzero numbers whose absolute value is less than some implementation-defined minimum. For the type `REAL`, that minimum will be at most `REAL'SAFE_SMALL`. For the subprograms defined by this International Standard, whenever the maximum relative error requirements permit a scalar result (or a scalar component or element of a composite result) whose absolute value is less than `REAL'SAFE_SMALL` and a prescribed result is not stipulated, the implementation may yield for that scalar result (or a scalar component or element of that composite result)

a) any value permitted by the maximum relative error requirements;

b) any nonzero value less than or equal to `REAL'SAFE_SMALL` in magnitude (and having the correct sign, unless the maximum relative error requirements permit values with either sign); or

c) zero.

NOTES

1 Whenever the behavior on underflow is as described in 12 b) or 12 c), the maximum relative error requirements are, in general, unachievable and are waived.

2 The rule permits an implementation to deliver a scalar result (or component or element of a composite result) violating the maximum relative error requirements for arguments for which the mathematical result (or component-part of the result) equals or slightly exceeds `REAL'SAFE_SMALL` in absolute value. Such arguments must necessarily be very close to an argument for which the mathematical result (or component-part of the result) is less than `REAL'SAFE_SMALL` in absolute value. In general, this is the best that can be expected from a portable implementation with a reasonable amount of effort.

13 Generic Complex Types Package

The generic package `GENERIC_COMPLEX_TYPES` defines operations and types for scalar complex arithmetic. One generic formal parameter, the floating-point type `REAL`, is defined for `GENERIC_COMPLEX_TYPES`. The corresponding generic actual parameter determines the precision of the arithmetic to be used in an instantiation of this generic package.

The Ada package specification for `GENERIC_COMPLEX_TYPES` is given in annex A.

13.1 Types

Two types are defined and exported by GENERIC_COMPLEX_TYPES. The type COMPLEX provides a cartesian representation of a complex number; it is declared as a record with two components which represent the real and imaginary parts. The type IMAGINARY is provided to represent a pure imaginary number; it is declared as a private type whose full type declaration reveals it to be derived from type REAL.

13.2 Constants

```
  i: constant IMAGINARY := 1.0;
  j: constant IMAGINARY := 1.0;
```

Each constant represents the imaginary unit value.

Each constant is exact.

13.3 COMPLEX selection, conversion and composition operations

```
function RE (X : COMPLEX) return REAL;
function IM (X : COMPLEX) return REAL;
```

Each function returns the specified cartesian component-part of X.

Each function is exact.

```
procedure SET_RE (X  : in out COMPLEX;
                  RE : in     REAL);
procedure SET_IM (X  : in out COMPLEX;
                  IM : in     REAL);
```

Each procedure resets the specified (cartesian) component of X; the other (cartesian) component is unchanged.

Each procedure is exact.

```
function "+" (LEFT  : REAL;
              RIGHT : IMAGINARY) return COMPLEX;
function "-" (LEFT  : REAL;
              RIGHT : IMAGINARY) return COMPLEX;
```

Each operation returns the COMPLEX result of applying the appropriate standard mathematical operation for arithmetic between real and imaginary numbers. This is also the standard mathematical operation for composing a complex number from real and imaginary numbers.

The real component-part of the result is exact. The imaginary component-part of the result shall satisfy the accuracy requirement of the appropriate unary operation for real arithmetic, as defined by Ada.

```
function "+" (LEFT  : IMAGINARY;
              RIGHT : REAL) return COMPLEX;
function "-" (LEFT  : IMAGINARY;
              RIGHT : REAL) return COMPLEX;
```

Each operation returns the COMPLEX result of applying the appropriate standard mathematical operation for arithmetic between real and imaginary numbers. This is also the standard mathematical operation for composing a complex number from real and imaginary numbers.

The real component-part of the result shall satisfy the accuracy requirement of the appropriate unary operation for real arithmetic, as defined by Ada. The imaginary component-part of the result is exact.