



INTERNATIONAL STANDARD ISO/IEC 8652:2012 TECHNICAL CORRIGENDUM 1

Published 2016-02-01

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION • МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ • ORGANISATION INTERNATIONALE DE NORMALISATION
INTERNATIONAL ELECTROTECHNICAL COMMISSION • МЕЖДУНАРОДНАЯ ЭЛЕКТРОТЕХНИЧЕСКАЯ КОМИССИЯ • COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

Programming languages — Ada

TECHNICAL CORRIGENDUM 1

Langages de programmation — Ada

RECTIFICATIF TECHNIQUE 1

Technical Corrigendum 1 to ISO/IEC 8652:2012 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: [Foreword - Supplementary information](#)

The committee responsible for this document is ISO/IEC JTC 1, *Information technology, SC22, Programming languages, their environments and system software interfaces*.

Technical Corrigendum 1 cancels and replaces those portions of International Standard ISO/IEC 8652:2012 as specified by the body of this corrigendum. Those portions of the International Standard not modified by this corrigendum remain in force.

Introduction

This corrigendum contains corrections to the Ada 2012 standard [ISO/IEC 8652:2012].

The corrigendum is organized by clauses corresponding to those in the Ada 2012 standard. These clauses include wording changes to the Ada 2012 standard. Subclause headings are given for each subclause that contains a wording change. Other subclauses are omitted. For each change, a reference to the defect report(s) that prompted the wording change is included in the form [8652/0000]. The defect reports have been developed by the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group to address specific questions about the Ada standard. Refer to the defect reports for details on the issues.

For each change, an *anchor* paragraph from the original Ada 2012 standard is given. New or revised text and instructions are given with each change. The anchor paragraph can be replaced or deleted, or text can be inserted before or after it. When a heading immediately precedes the anchor paragraph, any text inserted before the paragraph is intended to appear under the heading.

Typographical conventions:

Instructions about the text changes are in this font. The actual text changes are in the same fonts as the Ada 2012 standard - this font for text, this font for syntax, and this font for Ada source code.

Introduction

Of International Standard ISO/IEC 8652:2012. Modifications of this section of that standard are found here.

Replace paragraph 57.15: [8652/0117]

- The concept of assertions introduced in the 2005 edition is extended with the ability to specify preconditions and postconditions for subprograms, and invariants for private types. The concept of constraints in defining subtypes is supplemented with subtype predicates that enable subsets to be specified other than as simple ranges. These properties are all indicated using aspect specifications. See subclauses 3.2.4, 6.1.1, and 7.3.2.

by:

- The concept of assertions introduced in the 2005 edition is extended with the ability to specify preconditions and postconditions for subprograms, and invariants for private types and interfaces. The concept of constraints in defining subtypes is supplemented with subtype predicates that enable subsets to be specified other than as simple ranges. These properties are all indicated using aspect specifications. See subclauses 3.2.4, 6.1.1, and 7.3.2.

Replace paragraph 57.16: [8652/0117]

- New forms of expressions are introduced. These are if expressions, case expressions, quantified expressions, and expression functions. As well as being useful for programming in general by avoiding the introduction of unnecessary assignments, they are especially valuable in conditions and invariants since they avoid the need to introduce auxiliary functions. See subclauses 4.5.7, 4.5.8, and 6.8. Membership tests are also made more flexible. See subclauses 4.4 and 4.5.2.

by:

- New forms of expressions are introduced. These are if expressions, case expressions, quantified expressions, expression functions, and raise expressions. As well as being useful for programming in general by avoiding the introduction of unnecessary assignments, they are especially valuable in conditions and invariants since they avoid the need to introduce auxiliary functions. See subclauses 4.5.7, 4.5.8, 6.8, and 11.3. Membership tests are also made more flexible. See subclauses 4.4 and 4.5.2.

Section 1: General

1.1 Scope

Replace paragraph 3: [8652/0118]

The language provides rich support for real-time, concurrent programming, and includes facilities for multicore and multiprocessor programming. Errors can be signaled as exceptions and handled explicitly. The language also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, a predefined environment of standard packages is provided, including facilities for, among others, input-output, string manipulation, numeric elementary functions, and random number generation, and definition and use of containers.

by:

The language provides rich support for real-time, concurrent programming, and includes facilities for multicore and multiprocessor programming. Errors can be signaled as exceptions and handled explicitly. The language also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, a predefined environment of standard packages is provided, including facilities for, among others, input-output, string manipulation, numeric elementary functions, random number generation, and definition and use of containers.

1.1.2 Structure

Replace paragraph 24: [8652/0118]

Each section is divided into subclauses that have a common structure. Each clause and subclause first introduces its subject. After the introductory text, text is labeled with the following headings:

by:

Each clause is divided into subclauses that have a common structure. Each clause and subclause first introduces its subject. After the introductory text, text is labeled with the following headings:

Section 2: Lexical Elements

No changes in this clause.

Section 3: Declarations and Types

3.2.4 Subtype Predicates

Replace paragraph 4: [8652/0119; 8652/0120]

- For a (first) subtype defined by a derived type declaration, the predicates of the parent subtype and the progenitor subtypes apply.

by:

- For a (first) subtype defined by a type declaration, any predicates of parent or progenitor subtypes apply.

Delete paragraph 6: [8652/0119]

The *predicate* of a subtype consists of all predicate specifications that apply, and-ed together; if no predicate specifications apply, the predicate is True (in particular, the predicate of a base subtype is True).

Replace paragraph 12: [8652/0120]

- If a subtype is defined by a derived type declaration that does not include a predicate specification, then predicate checks are enabled for the subtype if and only if predicate checks are enabled for at least one of the parent subtype and the progenitor subtypes;

by:

- If a subtype is defined by a type declaration that does not include a predicate specification, then predicate checks are enabled for the subtype if and only if any predicate checks are enabled for parent or progenitor subtypes;

Insert after paragraph 14: [8652/0121]

- Otherwise, predicate checks are disabled for the given subtype.

the new paragraphs:

For a subtype with a directly-specified predicate aspect, the following additional language-defined aspect may be specified with an **aspect_specification** (see 13.1.1):

Predicate_Failure

This aspect shall be specified by an **expression**, which determines the action to be performed when a predicate check fails because a directly-specified predicate aspect of the subtype evaluates to False, as explained below.

Name Resolution Rules

The expected type for the Predicate_Failure **expression** is String.

Replace paragraph 17: [8652/0122]

- a membership test whose **simple_expression** is the current instance, and whose **membership_choice_list** meets the requirements for a static membership test (see 4.9);

by:

- a membership test whose **tested_simple_expression** is the current instance, and whose **membership_choice_list** meets the requirements for a static membership test (see 4.9);

Replace paragraph 20: [8652/0120]

- a call to a predefined boolean logical operator, where each operand is predicate-static;

by:

- a call to a predefined boolean operator **and**, **or**, **xor**, or **not**, where each operand is predicate-static;

Insert before paragraph 30: [8652/0119]

If predicate checks are enabled for a given subtype, then:

the new paragraphs:

If any of the above Legality Rules is violated in an instance of a generic unit, Program_Error is raised at the point of the violation.

To determine whether a value *satisfies the predicates* of a subtype *S*, the following tests are performed in the following order, until one of the tests fails, in which case the predicates are not satisfied and no further tests are performed, or all of the tests succeed, in which case the predicates are satisfied:

- the value is first tested to determine whether it satisfies any constraints or any null exclusion of *S*;
- then:
 - if *S* is a first subtype, the value is tested to determine whether it satisfies the predicates of the parent and progenitor subtypes (if any) of *S* (in an arbitrary order);
 - if *S* is defined by a `subtype_indication`, the value is tested to determine whether it satisfies the predicates of the subtype denoted by the `subtype_mark` of the `subtype_indication`;
- finally, if *S* is defined by a declaration to which one or more predicate specifications apply, the predicates are evaluated (in an arbitrary order) to test that all of them yield True for the given value.

Replace paragraph 31: [8652/0121; 8652/0119]

On every subtype conversion, the predicate of the target subtype is evaluated, and a check is performed that the predicate is True. This includes all parameter passing, except for certain parameters passed by reference, which are covered by the following rule: After normal completion and leaving of a subprogram, for each **in out** or **out** parameter that is passed by reference, the predicate of the subtype of the actual is evaluated, and a check is performed that the predicate is True. For an object created by an `object_declaration` with no explicit initialization `expression`, or by an uninitialized `allocator`, if any subcomponents have `default_expressions`, the predicate of the nominal subtype of the created object is evaluated, and a check is performed that the predicate is True. `Assertions.Assertion_Error` is raised if any of these checks fail.

by:

On every subtype conversion, a check is performed that the operand satisfies the predicates of the target subtype. This includes all parameter passing, except for certain parameters passed by reference, which are covered by the following rule: After normal completion and leaving of a subprogram, for each **in out** or **out** parameter that is passed by reference, a check is performed that the value of the parameter satisfies the predicates of the subtype of the actual. For an object created by an `object_declaration` with no explicit initialization `expression`, or by an uninitialized `allocator`, if any subcomponents have `default_expressions`, a check is performed that the value of the created object satisfies the predicates of the nominal subtype.

If any of the predicate checks fail, `Assertion_Error` is raised, unless the subtype whose directly-specified predicate aspect evaluated to False also has a directly-specified `Predicate_Failure` aspect. In that case, the specified `Predicate_Failure expression` is evaluated; if the evaluation of the `Predicate_Failure expression` propagates an exception occurrence, then this occurrence is propagated for the failure of the predicate check; otherwise, `Assertion_Error` is raised, with an associated message string defined by the value of the `Predicate_Failure expression`. In the absence of such a `Predicate_Failure` aspect, an implementation-defined message string is associated with the `Assertion_Error` exception.

Delete paragraph 32: [8652/0119]

A value *satisfies* a predicate if the predicate is True for that value.

Delete paragraph 33: [8652/0119]

If any of the above Legality Rules is violated in an instance of a generic unit, `Program_Error` is raised at the point of the violation.

Insert after paragraph 35: [8652/0121; 8652/0119]

6 A `Static_Predicate`, like a constraint, always remains True for all objects of the subtype, except in the case of uninitialized variables and other invalid values. A `Dynamic_Predicate`, on the other hand, is checked as specified above, but can become False at other times. For example, the predicate of a record subtype is not checked when a subcomponent is modified.

the new paragraphs:

7 No predicates apply to the base subtype of a scalar type; every value of a scalar type *T* is considered to satisfy the predicates of *T*'Base.

8 `Predicate_Failure expressions` are never evaluated during the evaluation of a membership test (see 4.5.2) or Valid attribute (see 13.9.2).

9 A Predicate_Failure expression can be a raise_expression (see 11.3).

Examples

```
subtype Basic_Letter is Character -- See A.3.2 for "basic letter".
with Static_Predicate => Basic_Letter in 'A'..'Z' | 'a'..'z' | 'Æ' | 'æ' | 'Ð' | 'ð'
| 'Þ' | 'þ' | 'ß';

subtype Even_Integer is Integer
with Dynamic_Predicate => Even_Integer mod 2 = 0,
    Predicate_Failure => "Even_Integer must be a multiple of 2";
```

Text_IO (see A.10.1) could have used predicates to describe some common exceptional conditions as follows:

```
with Ada.IO_Exceptions;
package Ada.Text_IO is

    type File_Type is limited private;

    subtype Open_File_Type is File_Type
    with Dynamic_Predicate => Is_Open (Open_File_Type),
        Predicate_Failure => raise Status_Error with "File not open";

    subtype Input_File_Type is Open_File_Type
    with Dynamic_Predicate => Mode (Input_File_Type) = In_File,
        Predicate_Failure => raise Mode_Error with "Cannot read file: " &
            Name (Input_File_Type);

    subtype Output_File_Type is Open_File_Type
    with Dynamic_Predicate => Mode (Output_File_Type) /= In_File,
        Predicate_Failure => raise Mode_Error with "Cannot write file: " &
            Name (Output_File_Type);

    ...

    function Mode (File : in Open_File_Type) return File_Mode;
    function Name (File : in Open_File_Type) return String;
    function Form (File : in Open_File_Type) return String;

    ...

    procedure Get (File : in Input_File_Type; Item : out Character);
    procedure Put (File : in Output_File_Type; Item : in Character);

    ...

    -- Similarly for all of the other input and output subprograms.
```

3.5 Scalar Types

Insert after paragraph 55: [8652/0123]

For the evaluation of a call on S'Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Image for a value of the type), the result is the corresponding enumeration value; otherwise, Constraint_Error is raised. For a numeric subtype S, the evaluation of a call on S'Value with Arg of type String is equivalent to a call on S'Wide_Wide_Value for a corresponding Arg of type Wide_Wide_String.

the new paragraphs:

For a prefix X that denotes an object of a scalar type (after any implicit dereference), the following attributes are defined:

X'Wide_Wide_Image

X'Wide_Wide_Image denotes the result of calling function S'Wide_Wide_Image with Arg being X, where S is the nominal subtype of X.

X'Wide_Image

X'Wide_Image denotes the result of calling function S'Wide_Image with Arg being X, where S is the nominal subtype of X.

X'Image

X'Image denotes the result of calling function S'Image with *Arg* being X, where S is the nominal subtype of X.

3.5.5 Operations of Discrete Types

Replace paragraph 7.1: [8652/0119]

For every static discrete subtype S for which there exists at least one value belonging to S that satisfies any predicate of S, the following attributes are defined:

by:

For every static discrete subtype S for which there exists at least one value belonging to S that satisfies the predicates of S, the following attributes are defined:

Replace paragraph 7.2: [8652/0119]

S'First_Valid

S'First_Valid denotes the smallest value that belongs to S and satisfies the predicate of S. The value of this attribute is of the type of S.

by:

S'First_Valid

S'First_Valid denotes the smallest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S.

Replace paragraph 7.3: [8652/0119]

S'Last_Valid

S'Last_Valid denotes the largest value that belongs to S and satisfies the predicate of S. The value of this attribute is of the type of S.

by:

S'Last_Valid

S'Last_Valid denotes the largest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S.

3.5.9 Fixed Point Types

Replace paragraph 5: [8652/0124]

digits_constraint ::=

digits static_expression [range_constraint]

by:

digits_constraint ::=

digits static_simple_expression [range_constraint]

Insert after paragraph 6: [8652/0125]

For a type defined by a **fixed_point_definition**, the *delta* of the type is specified by the value of the **expression** given after the reserved word **delta**; this **expression** is expected to be of any real type. For a type defined by a **decimal_fixed_point_definition** (a *decimal* fixed point type), the number of significant decimal digits for its first subtype (the *digits* of the first subtype) is specified by the **expression** given after the reserved word **digits**; this **expression** is expected to be of any integer type.

the new paragraph:

The **simple_expression** of a **digits_constraint** is expected to be of any integer type.

Replace paragraph 18: [8652/0124]

For a **digits_constraint** on a decimal fixed point subtype with a given *delta*, if it does not have a **range_constraint**, then it specifies an implicit range $-(10^{**D}-1)*delta .. +(10^{**D}-1)*delta$, where *D* is the value of the **expression**. A **digits_constraint** is *compatible* with a decimal fixed point subtype if the value of the **expression** is no greater than the *digits* of the subtype, and if it specifies (explicitly or implicitly) a range that is compatible with the subtype.

by:

For a `digits_constraint` on a decimal fixed point subtype with a given `delta`, if it does not have a `range_constraint`, then it specifies an implicit range $-(10^{**D}-1)*delta .. +(10^{**D}-1)*delta$, where `D` is the value of the `simple_expression`. A `digits_constraint` is *compatible* with a decimal fixed point subtype if the value of the `simple_expression` is no greater than the `digits` of the subtype, and if it specifies (explicitly or implicitly) a range that is compatible with the subtype.

Replace paragraph 19: [8652/0124]

The elaboration of a `digits_constraint` consists of the elaboration of the `range_constraint`, if any. If a `range_constraint` is given, a check is made that the bounds of the range are both in the range $-(10^{**D}-1)*delta .. +(10^{**D}-1)*delta$, where `D` is the value of the (static) `expression` given after the reserved word **digits**. If this check fails, `Constraint_Error` is raised.

by:

The elaboration of a `digits_constraint` consists of the elaboration of the `range_constraint`, if any. If a `range_constraint` is given, a check is made that the bounds of the range are both in the range $-(10^{**D}-1)*delta .. +(10^{**D}-1)*delta$, where `D` is the value of the (static) `simple_expression` given after the reserved word **digits**. If this check fails, `Constraint_Error` is raised.

3.8.1 Variant Parts and Discrete Choices

Replace paragraph 10.1: [8652/0119]

- A `discrete_choice` that is a `subtype_indication` covers all values (possibly none) that belong to the subtype and that satisfy the static predicate of the subtype (see 3.2.4).

by:

- A `discrete_choice` that is a `subtype_indication` covers all values (possibly none) that belong to the subtype and that satisfy the static predicates of the subtype (see 3.2.4).

Replace paragraph 15: [8652/0119]

- If the discriminant is of a static constrained scalar subtype then, except within an instance of a generic unit, each non-**others** `discrete_choice` shall cover only values in that subtype that satisfy its predicate, and each value of that subtype that satisfies its predicate shall be covered by some `discrete_choice` (either explicitly or by **others**);

by:

- If the discriminant is of a static constrained scalar subtype then, except within an instance of a generic unit, each non-**others** `discrete_choice` shall cover only values in that subtype that satisfy its predicates, and each value of that subtype that satisfies its predicates shall be covered by some `discrete_choice` (either explicitly or by **others**);

3.9 Tagged Types and Type Extensions

Replace paragraph 12.4: [8652/0118]

The function `Parent_Tag` returns the tag of the parent type of the type whose tag is `T`. If the type does not have a parent type (that is, it was not declared by a `derived_type_declaration`), then `No_Tag` is returned.

by:

The function `Parent_Tag` returns the tag of the parent type of the type whose tag is `T`. If the type does not have a parent type (that is, it was not defined by a `derived_type_definition`), then `No_Tag` is returned.

3.9.3 Abstract Types and Subprograms

Replace paragraph 6: [8652/0126]

- Otherwise, the subprogram shall be overridden with a nonabstract subprogram or, in the case of a private extension inheriting a function with a controlling result, have a full type that is a null extension; for a type declared in the visible part of a package, the overriding may be either in the visible or the private part. Such a subprogram is said to *require overriding*. However, if the type is a generic formal type, the subprogram need not be overridden for the formal type itself; a nonabstract version will necessarily be provided by the actual type.

by:

- Otherwise, the subprogram shall be overridden with a nonabstract subprogram or, in the case of a private extension inheriting a nonabstract function with a controlling result, have a full type that is a null extension; for a type declared in the visible part of a package, the overriding may be either in the visible or the private part. Such a subprogram is said to *require overriding*. However, if the type is a generic formal type, the subprogram need not be overridden for the formal type itself; a nonabstract version will necessarily be provided by the actual type.

3.10 Access Types

Replace paragraph 22: [8652/0118]

```
type Peripheral_Ref is not null access Peripheral; -- see 3.8.1
type Binop_Ptr is access all Binary_Operation'Class;
-- general access-to-class-wide, see 3.9.1
```

by:

```
type Frame is access Matrix; -- see 3.6
type Peripheral_Ref is not null access Peripheral; -- see 3.8.1
type Binop_Ptr is access all Binary_Operation'Class;
-- general access-to-class-wide, see 3.9.1
```

3.10.1 Incomplete Type Declarations

Replace paragraph 2.1: [8652/0127]

An *incomplete_type_declaration* declares an *incomplete view* of a type and its first subtype; the first subtype is unconstrained if a *discriminant_part* appears. If the *incomplete_type_declaration* includes the reserved word **tagged**, it declares a *tagged incomplete view*. An incomplete view of a type is a limited view of the type (see 7.5).

by:

An *incomplete_type_declaration* declares an *incomplete view* of a type and its first subtype; the first subtype is unconstrained if a *discriminant_part* appears. If the *incomplete_type_declaration* includes the reserved word **tagged**, it declares a *tagged incomplete view*. If *T* denotes a tagged incomplete view, then *T'Class* denotes a tagged incomplete view. An incomplete view of a type is a limited view of the type (see 7.5).

3.10.2 Operations of Access Types

Replace paragraph 7: [8652/0128; 8652/0129]

- An entity or view defined by a declaration and created as part of its elaboration has the same accessibility level as the innermost master of the declaration except in the cases of renaming and derived access types described below. Other than for an explicitly aliased parameter, a formal parameter of a callable entity has the same accessibility level as the master representing the invocation of the entity.

by:

- An entity or view defined by a declaration and created as part of its elaboration has the same accessibility level as the innermost master of the declaration except in the cases of renaming and derived access types described below. Other than for an explicitly aliased parameter of a function or generic function, a formal parameter of a callable entity has the same accessibility level as the master representing the invocation of the entity.

Replace paragraph 10: [8652/0130]

- The accessibility level of an **aggregate** that is used (in its entirety) to directly initialize part of an object is that of the object being initialized. In other contexts, the accessibility level of an **aggregate** is that of the innermost master that evaluates the **aggregate**.

by:

- The accessibility level of an **aggregate** that is used (in its entirety) to directly initialize part of an object is that of the object being initialized. In other contexts, the accessibility level of an **aggregate** is that of the innermost master that evaluates the **aggregate**. Corresponding rules apply to a value conversion (see 4.6).

Insert after paragraph 13.1: [8652/0131]

- The accessibility level of the anonymous access type of an access parameter specifying an access-to-subprogram type is deeper than that of any master; all such anonymous access types have this same level.

the new paragraph:

- The accessibility level of the anonymous access subtype defined by a `return_subtype_indication` that is an `access_definition` (see 6.5) is that of the result subtype of the enclosing function.

Replace paragraph 19.2: [8652/0129; 8652/0132]

- Inside a return statement that applies to a function *F*, when determining whether the accessibility level of an explicitly aliased parameter of *F* is statically deeper than the level of the return object of *F*, the level of the return object is considered to be the same as that of the level of the explicitly aliased parameter; for statically comparing with the level of other entities, an explicitly aliased parameter of *F* is considered to have the accessibility level of the body of *F*.

by:

- Inside a return statement that applies to a function or generic function *F*, or the return expression of an expression function *F*, when determining whether the accessibility level of an explicitly aliased parameter of *F* is statically deeper than the level of the return object of *F*, the level of the return object is considered to be the same as that of the level of the explicitly aliased parameter; for statically comparing with the level of other entities, an explicitly aliased parameter of *F* is considered to have the accessibility level of the body of *F*.

Replace paragraph 19.3: [8652/0129; 8652/0132]

- For determining whether a level is statically deeper than the level of the anonymous access type of an access result of a function, when within a return statement that applies to the function, the level of the master of the call is presumed to be the same as that of the level of the master that elaborated the function body.

by:

- For determining whether a level is statically deeper than the level of the anonymous access type of an access result of a function or generic function *F*, when within a return statement that applies to *F* or the return expression of expression function *F*, the level of the master of the call is presumed to be the same as that of the level of the master that elaborated the body of *F*.

Replace paragraph 27.2: [8652/0133]

- *D* shall be discriminated in its full view and unconstrained in any partial view, and the designated subtype of *A* shall be unconstrained. For the purposes of determining within a generic body whether *D* is unconstrained in any partial view, a discriminated subtype is considered to have a constrained partial view if it is a descendant of an untagged generic formal private or derived type.

by:

- *D* shall be discriminated in its full view and unconstrained in any partial view, and the designated subtype of *A* shall be unconstrained.

Section 4: Names and Expressions

4.1.4 Attributes

Replace paragraph 9: [8652/0134; 8652/0125]

An *attribute_reference* denotes a value, an object, a subprogram, or some other kind of program entity. For an *attribute_reference* that denotes a value or an object, if its type is scalar, then its nominal subtype is the base subtype of the type; if its type is tagged, its nominal subtype is the first subtype of the type; otherwise, its nominal subtype is a subtype of the type without any constraint or *null_exclusion*. Similarly, unless explicitly specified otherwise, for an *attribute_reference* that denotes a function, when its result type is scalar, its result subtype is the base subtype of the type, when its result type is tagged, the result subtype is the first subtype of the type, and when the result type is some other type, the result subtype is a subtype of the type without any constraint or *null_exclusion*.

by:

An *attribute_reference* denotes a value, an object, a subprogram, or some other kind of program entity. Unless explicitly specified otherwise, for an *attribute_reference* that denotes a value or an object, if its type is scalar, then its nominal subtype is the base subtype of the type; if its type is tagged, its nominal subtype is the first subtype of the type; otherwise, its nominal subtype is a subtype of the type without any constraint, *null_exclusion*, or predicate. Similarly, unless explicitly specified otherwise, for an *attribute_reference* that denotes a function, when its result type is scalar, its result subtype is the base subtype of the type, when its result type is tagged, the result subtype is the first subtype of the type, and when the result type is some other type, the result subtype is a subtype of the type without any constraint, *null_exclusion*, or predicate.

4.1.5 User-defined References

Insert before paragraph 6: [8652/0135]

A *generalized_reference* denotes a view equivalent to that of a dereference of the reference discriminant of the reference object.

the new paragraph:

The *Implicit_Dereference* aspect is nonoverridable (see 13.1.1).

4.1.6 User-defined Indexing

Replace paragraph 4: [8652/0136]

These aspects are inherited by descendants of *T* (including the class-wide type *TClass*). The aspects shall not be overridden, but the functions they denote may be.

by:

These aspects are inherited by descendants of *T* (including the class-wide type *TClass*).

Insert after paragraph 5: [8652/0135]

An *indexable container type* is (a view of) a tagged type with at least one of the aspects *Constant_Indexing* or *Variable_Indexing* specified. An *indexable container object* is an object of an indexable container type. A *generalized_indexing* is a name that denotes the result of calling a function named by a *Constant_Indexing* or *Variable_Indexing* aspect.

the new paragraph:

The *Constant_Indexing* and *Variable_Indexing* aspects are nonoverridable (see 13.1.1).

Delete paragraph 6: [8652/0135]

The *Constant_Indexing* or *Variable_Indexing* aspect shall not be specified:

Delete paragraph 7: [8652/0135]

- on a derived type if the parent type has the corresponding aspect specified or inherited; or

Delete paragraph 8: [8652/0135]

- on a *full_type_declaration* if the type has a tagged partial view.

Delete paragraph 9: [8652/0135]

In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

Insert after paragraph 17: [8652/0136]

When a `generalized_indexing` is interpreted as a constant (or variable) indexing, it is equivalent to a call on a prefixed view of one of the functions named by the `Constant_Indexing` (or `Variable_Indexing`) aspect of the type of the `indexable_container_object_prefix` with the given `actual_parameter_part`, and with the `indexable_container_object_prefix` as the prefix of the prefixed view.

the new paragraph:

NOTES

- 6 The `Constant_Indexing` and `Variable_Indexing` aspects cannot be redefined when inherited for a derived type, but the functions that they denote can be modified by overriding or overloading.

4.3.1 Record Aggregates**Replace paragraph 16: [8652/0137]**

Each `record_component_association` other than an **others** choice with a $\triangleleft$ shall have at least one associated component, and each needed component shall be associated with exactly one `record_component_association`. If a `record_component_association` with an `expression` has two or more associated components, all of them shall be of the same type, or all of them shall be of anonymous access types whose subtypes statically match.

by:

Each `record_component_association` other than an **others** choice with a $\triangleleft$ shall have at least one associated component, and each needed component shall be associated with exactly one `record_component_association`. If a `record_component_association` with an `expression` has two or more associated components, all of them shall be of the same type, or all of them shall be of anonymous access types whose subtypes statically match. In addition, Legality Rules are enforced separately for each associated component.

4.3.3 Array Aggregates**Replace paragraph 11: [8652/0132]**

For an `explicit_actual_parameter`, an `explicit_generic_actual_parameter`, the `expression` of a return statement, the initialization expression in an `object_declaration`, or a `default_expression` (for a parameter or a component), when the nominal subtype of the corresponding formal parameter, generic formal parameter, function return object, object, or component is a constrained array subtype, the applicable index constraint is the constraint of the subtype;

by:

For an `explicit_actual_parameter`, an `explicit_generic_actual_parameter`, the `expression` of a return statement, the return expression of an expression function, the initialization expression in an `object_declaration`, or a `default_expression` (for a parameter or a component), when the nominal subtype of the corresponding formal parameter, generic formal parameter, function return object, expression function return object, object, or component is a constrained array subtype, the applicable index constraint is the constraint of the subtype;

Replace paragraph 23.1: [8652/0138]

Each expression in an `array_component_association` defines the value for the associated component(s). For an `array_component_association` with $\triangleleft$, the associated component(s) are initialized by default as for a stand-alone object of the component subtype (see 3.3.1).

by:

Each expression in an `array_component_association` defines the value for the associated component(s). For an `array_component_association` with $\triangleleft$, the associated component(s) are initialized to the `Default_Component_Value` of the array type if this aspect has been specified for the array type; otherwise, they are initialized by default as for a stand-alone object of the component subtype (see 3.3.1).

4.4 Expressions

Replace paragraph 3: [8652/0139; 8652/0122]

```
relation ::=
    simple_expression [relational_operator simple_expression]
    | simple_expression [not] in membership_choice_list
```

by:

```
relation ::=
    simple_expression [relational_operator simple_expression]
    | tested_simple_expression [not] in membership_choice_list
    | raise_expression
```

Replace paragraph 3.2: [8652/0122]

```
membership_choice ::= choice_expression | range | subtype_mark
```

by:

```
membership_choice ::= choice_simple_expression | range | subtype_mark
```

4.5.2 Relational Operators and Membership Tests

Replace paragraph 3.1: [8652/0122]

If the tested type is tagged, then the *simple_expression* shall resolve to be of a type that is convertible (see 4.6) to the tested type; if untagged, the expected type for the *simple_expression* is the tested type. The expected type of a *choice_expression* in a *membership_choice*, and of a *simple_expression* of a *range* in a *membership_choice*, is the tested type of the membership operation.

by:

If the tested type is tagged, then the *tested_simple_expression* shall resolve to be of a type that is convertible (see 4.6) to the tested type; if untagged, the expected type for the *tested_simple_expression* is the tested type. The expected type of a *choice_simple_expression* in a *membership_choice*, and of a *simple_expression* of a *range* in a *membership_choice*, is the tested type of the membership operation.

Replace paragraph 4: [8652/0122]

For a membership test, if the *simple_expression* is of a tagged class-wide type, then the tested type shall be (visibly) tagged.

by:

For a membership test, if the *tested_simple_expression* is of a tagged class-wide type, then the tested type shall be (visibly) tagged.

Replace paragraph 4.1: [8652/0122]

If a membership test includes one or more *choice_expressions* and the tested type of the membership test is limited, then the tested type of the membership test shall have a visible primitive equality operator.

by:

If a membership test includes one or more *choice_simple_expressions* and the tested type of the membership test is limited, then the tested type of the membership test shall have a visible primitive equality operator.

Replace paragraph 9.8: [8652/0140]

If the profile of an explicitly declared primitive equality operator of an untagged record type is type conformant with that of the corresponding predefined equality operator, the declaration shall occur before the type is frozen. In addition, if the untagged record type has a nonlimited partial view, then the declaration shall occur in the visible part of the enclosing package. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

by:

If the profile of an explicitly declared primitive equality operator of an untagged record type is type conformant with that of the corresponding predefined equality operator, the declaration shall occur before the type is frozen. In addition to the

places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

Replace paragraph 27: [8652/0122]

For the evaluation of a membership test using **in** whose `membership_choice_list` has a single `membership_choice`, the `simple_expression` and the `membership_choice` are evaluated in an arbitrary order; the result is the result of the individual membership test for the `membership_choice`.

by:

For the evaluation of a membership test using **in** whose `membership_choice_list` has a single `membership_choice`, the `tested_simple_expression` and the `membership_choice` are evaluated in an arbitrary order; the result is the result of the individual membership test for the `membership_choice`.

Replace paragraph 27.1: [8652/0122]

For the evaluation of a membership test using **in** whose `membership_choice_list` has more than one `membership_choice`, the `simple_expression` of the membership test is evaluated first and the result of the operation is equivalent to that of a sequence consisting of an individual membership test on each `membership_choice` combined with the short-circuit control form **or else**.

by:

For the evaluation of a membership test using **in** whose `membership_choice_list` has more than one `membership_choice`, the `tested_simple_expression` of the membership test is evaluated first and the result of the operation is equivalent to that of a sequence consisting of an individual membership test on each `membership_choice` combined with the short-circuit control form **or else**.

Replace paragraph 28.1: [8652/0122]

- The `membership_choice` is a `choice_expression`, and the `simple_expression` is equal to the value of the `membership_choice`. If the tested type is a record type or a limited type, the test uses the primitive equality for the type; otherwise, the test uses predefined equality.

by:

- The `membership_choice` is a `choice_simple_expression`, and the `tested_simple_expression` is equal to the value of the `membership_choice`. If the tested type is a record type or a limited type, the test uses the primitive equality for the type; otherwise, the test uses predefined equality.

Replace paragraph 28.2: [8652/0122]

- The `membership_choice` is a range and the value of the `simple_expression` belongs to the given range.

by:

- The `membership_choice` is a range and the value of the `tested_simple_expression` belongs to the given range.

Replace paragraph 29: [8652/0122; 8652/0119]

- The `membership_choice` is a `subtype_mark`, the tested type is scalar, the value of the `simple_expression` belongs to the range of the named subtype, and the predicate of the named subtype evaluates to True.

by:

- The `membership_choice` is a `subtype_mark`, the tested type is scalar, the value of the `tested_simple_expression` belongs to the range of the named subtype, and the value satisfies the predicates of the named subtype.

Replace paragraph 30: [8652/0122; 8652/0119]

- The `membership_choice` is a `subtype_mark`, the tested type is not scalar, the value of the `simple_expression` satisfies any constraints of the named subtype, the predicate of the named subtype evaluates to True, and:

by:

- The `membership_choice` is a `subtype_mark`, the tested type is not scalar, the value of the `tested_simple_expression` satisfies any constraints of the named subtype, the value satisfies the predicates of the named subtype, and:

Replace paragraph 30.1: [8652/0122]

if the type of the `simple_expression` is class-wide, the value has a tag that identifies a type covered by the tested type;

by:

if the type of the `tested_simple_expression` is class-wide, the value has a tag that identifies a type covered by the tested type;

Replace paragraph 30.2: [8652/0122]

- if the tested type is an access type and the named subtype excludes null, the value of the `simple_expression` is not null;

by:

- if the tested type is an access type and the named subtype excludes null, the value of the `tested_simple_expression` is not null;

Replace paragraph 30.3: [8652/0122]

- if the tested type is a general access-to-object type, the type of the `simple_expression` is convertible to the tested type and its accessibility level is no deeper than that of the tested type; further, if the designated type of the tested type is tagged and the `simple_expression` is nonnull, the tag of the object designated by the value of the `simple_expression` is covered by the designated type of the tested type.

by:

- if the tested type is a general access-to-object type, the type of the `tested_simple_expression` is convertible to the tested type and its accessibility level is no deeper than that of the tested type; further, if the designated type of the tested type is tagged and the `tested_simple_expression` is nonnull, the tag of the object designated by the value of the `tested_simple_expression` is covered by the designated type of the tested type.

4.5.8 Quantified Expressions

Insert before paragraph 1: [8652/0141]

```
quantified_expression ::= for quantifier loop_parameter_specification => predicate
| for quantifier iterator_specification => predicate
```

the new paragraph:

Quantified expressions provide a way to write universally and existentially quantified predicates over containers and arrays.

Replace paragraph 6: [8652/0141]

For the evaluation of a `quantified_expression`, the `loop_parameter_specification` or `iterator_specification` is first elaborated. The evaluation of a `quantified_expression` then evaluates the `predicate` for each value of the loop parameter. These values are examined in the order specified by the `loop_parameter_specification` (see 5.5) or `iterator_specification` (see 5.5.2).

by:

For the evaluation of a `quantified_expression`, the `loop_parameter_specification` or `iterator_specification` is first elaborated. The evaluation of a `quantified_expression` then evaluates the `predicate` for the values of the loop parameter in the order specified by the `loop_parameter_specification` (see 5.5) or `iterator_specification` (see 5.5.2).

Replace paragraph 8: [8652/0141]

- If the `quantifier` is **all**, the expression is True if the evaluation of the `predicate` yields True for each value of the loop parameter. It is False otherwise. Evaluation of the `quantified_expression` stops when all values of the domain have been examined, or when the `predicate` yields False for a given value. Any exception raised by evaluation of the `predicate` is propagated.

by:

- If the `quantifier` is **all**, the expression is False if the evaluation of any `predicate` yields False; evaluation of the `quantified_expression` stops at that point. Otherwise (every `predicate` has been evaluated and yielded True), the expression is True. Any exception raised by evaluation of the `predicate` is propagated.

Replace paragraph 9: [8652/0141]

- If the quantifier is **some**, the expression is True if the evaluation of the **predicate** yields True for some value of the loop parameter. It is False otherwise. Evaluation of the **quantified_expression** stops when all values of the domain have been examined, or when the **predicate** yields True for a given value. Any exception raised by evaluation of the **predicate** is propagated.

by:

- If the quantifier is **some**, the expression is True if the evaluation of any **predicate** yields True; evaluation of the **quantified_expression** stops at that point. Otherwise (every **predicate** has been evaluated and yielded False), the expression is False. Any exception raised by evaluation of the **predicate** is propagated.

4.6 Type Conversions

Replace paragraph 24.17: [8652/0130]

- The accessibility level of the operand type shall not be statically deeper than that of the target type, unless the target type is an anonymous access type of a stand-alone object. If the target type is that of such a stand-alone object, the accessibility level of the operand type shall not be statically deeper than that of the declaration of the stand-alone object. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

by:

- The accessibility level of the operand type shall not be statically deeper than that of the target type, unless the target type is an anonymous access type of a stand-alone object. If the target type is that of such a stand-alone object, the accessibility level of the operand type shall not be statically deeper than that of the declaration of the stand-alone object.

Replace paragraph 24.21: [8652/0130]

- The accessibility level of the operand type shall not be statically deeper than that of the target type. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. If the operand type is declared within a generic body, the target type shall be declared within the generic body.

by:

- The accessibility level of the operand type shall not be statically deeper than that of the target type. If the operand type is declared within a generic body, the target type shall be declared within the generic body.

In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

Replace paragraph 51: [8652/0119]

After conversion of the value to the target type, if the target subtype is constrained, a check is performed that the value satisfies this constraint. If the target subtype excludes null, then a check is made that the value is not null. If predicate checks are enabled for the target subtype (see 3.2.4), a check is performed that the predicate of the target subtype is satisfied for the value.

by:

After conversion of the value to the target type, if the target subtype is constrained, a check is performed that the value satisfies this constraint. If the target subtype excludes null, then a check is made that the value is not null. If predicate checks are enabled for the target subtype (see 3.2.4), a check is performed that the value satisfies the predicates of the target subtype.

Replace paragraph 56: [8652/0142]

- Reading the value of the view yields the result of converting the value of the operand object to the target subtype (which might raise **Constraint_Error**), except if the object is of an access type and the view conversion is passed as an **out** parameter; in this latter case, the value of the operand object is used to initialize the formal parameter without checking against any constraint of the target subtype (see 6.4.1).

by:

- Reading the value of the view yields the result of converting the value of the operand object to the target subtype (which might raise `Constraint_Error`), except if the object is of an elementary type and the view conversion is passed as an `out` parameter; in this latter case, the value of the operand object may be used to initialize the formal parameter without checking against any constraint of the target subtype (as described more precisely in 6.4.1).

Replace paragraph 57: [8652/0143]

If an `Accessibility_Check` fails, `Program_Error` is raised. If a predicate check fails, `Assertions.Assertion_Error` is raised. Any other check associated with a conversion raises `Constraint_Error` if it fails.

by:

If an `Accessibility_Check` fails, `Program_Error` is raised. If a predicate check fails, the effect is as defined in subclause 3.2.4, "Subtype Predicates". Any other check associated with a conversion raises `Constraint_Error` if it fails.

Insert after paragraph 58: [8652/0130]

Conversion to a type is the same as conversion to an unconstrained subtype of the type.

the new paragraphs:

Evaluation of a value conversion of a composite type either creates a new anonymous object (similar to the object created by the evaluation of an `aggregate` or a function call) or yields a new view of the operand object without creating a new object:

- If the target type is a by-reference type and there is a type that is an ancestor of both the target type and the operand type then no new object is created;
- If the target type is an array type having aliased components and the operand type is an array type having unaliased components, then a new object is created;
- Otherwise, it is unspecified whether a new object is created.

If a new object is created, then the initialization of that object is an assignment operation.

4.7 Qualified Expressions

Replace paragraph 4: [8652/0144]

The evaluation of a `qualified_expression` evaluates the operand (and if of a universal type, converts it to the type determined by the `subtype_mark`) and checks that its value belongs to the subtype denoted by the `subtype_mark`. The exception `Constraint_Error` is raised if this check fails.

by:

The evaluation of a `qualified_expression` evaluates the operand (and if of a universal type, converts it to the type determined by the `subtype_mark`) and checks that its value belongs to the subtype denoted by the `subtype_mark`. The exception `Constraint_Error` is raised if this check fails. Furthermore, if predicate checks are enabled for the subtype denoted by the `subtype_mark`, a check is performed as defined in subclause 3.2.4, "Subtype Predicates" that the value satisfies the predicates of the subtype.

4.9 Static Expressions and Static Subtypes

Replace paragraph 11: [8652/0122]

- a membership test whose `simple_expression` is a static expression, and whose `membership_choice_list` consists only of `membership_choices` that are either static `choice_expressions`, static ranges, or `subtype_marks` that denote a static (scalar or string) subtype;

by:

- a membership test whose `tested_simple_expression` is a static expression, and whose `membership_choice_list` consists only of `membership_choices` that are either static `choice_simple_expressions`, static ranges, or `subtype_marks` that denote a static (scalar or string) subtype;

Replace paragraph 32.6: [8652/0122]

- a *choice_expression* (or a *simple_expression* of a *range* that occurs as a *membership_choice* of a *membership_choice_list*) of a static membership test that is preceded in the enclosing *membership_choice_list* by another item whose individual membership test (see 4.5.2) statically yields True.

by:

- a *choice_simple_expression* (or a *simple_expression* of a *range* that occurs as a *membership_choice* of a *membership_choice_list*) of a static membership test that is preceded in the enclosing *membership_choice_list* by another item whose individual membership test (see 4.5.2) statically yields True.

4.9.1 Statically Matching Constraints and Subtypes

Replace paragraph 10: [8652/0119]

- both subtypes are static, every value that satisfies the predicate of *S1* also satisfies the predicate of *S2*, and it is not the case that both types each have at least one applicable predicate specification, predicate checks are enabled (see 11.4.2) for *S2*, and predicate checks are not enabled for *S1*.

by:

- both subtypes are static, every value that satisfies the predicates of *S1* also satisfies the predicates of *S2*, and it is not the case that both types each have at least one applicable predicate specification, predicate checks are enabled (see 11.4.2) for *S2*, and predicate checks are not enabled for *S1*.

Section 5: Statements

5.2 Assignment Statements

Replace paragraph 20: [8652/0118]

```
Writer := (Status => Open, Unit => Printer, Line_Count => 60); -- see 3.8.1
Next_Car.all := (72074, null); -- see 3.10.1
```

by:

```
Writer := (Status => Open, Unit => Printer, Line_Count => 60); -- see 3.8.1
Next.all := (72074, null, Head); -- see 3.10.1
```

5.4 Case Statements

Replace paragraph 7: [8652/0119]

- If the *selecting_expression* is a name (including a *type_conversion*, *qualified_expression*, or *function_call*) having a static and constrained nominal subtype, then each non-**others** *discrete_choice* shall cover only values in that subtype that satisfy its predicate (see 3.2.4), and each value of that subtype that satisfies its predicate shall be covered by some *discrete_choice* (either explicitly or by **others**).

by:

- If the *selecting_expression* is a name (including a *type_conversion*, *qualified_expression*, or *function_call*) having a static and constrained nominal subtype, then each non-**others** *discrete_choice* shall cover only values in that subtype that satisfy its predicates (see 3.2.4), and each value of that subtype that satisfies its predicates shall be covered by some *discrete_choice* (either explicitly or by **others**).

5.5 Loop Statements

Replace paragraph 9: [8652/0119]

For the execution of a *loop_statement* with the *iteration_scheme* being **for** *loop_parameter_specification*, the *loop_parameter_specification* is first elaborated. This elaboration creates the loop parameter and elaborates the *discrete_subtype_definition*. If the *discrete_subtype_definition* defines a subtype with a null range, the execution of the *loop_statement* is complete. Otherwise, the *sequence_of_statements* is executed once for each value of the discrete subtype defined by the *discrete_subtype_definition* that satisfies the predicate of the subtype (or until the loop is left as a consequence of a transfer of control). Prior to each such iteration, the corresponding value of the discrete subtype is assigned to the loop parameter. These values are assigned in increasing order unless the reserved word **reverse** is present, in which case the values are assigned in decreasing order.

by:

For the execution of a *loop_statement* with the *iteration_scheme* being **for** *loop_parameter_specification*, the *loop_parameter_specification* is first elaborated. This elaboration creates the loop parameter and elaborates the *discrete_subtype_definition*. If the *discrete_subtype_definition* defines a subtype with a null range, the execution of the *loop_statement* is complete. Otherwise, the *sequence_of_statements* is executed once for each value of the discrete subtype defined by the *discrete_subtype_definition* that satisfies the predicates of the subtype (or until the loop is left as a consequence of a transfer of control). Prior to each such iteration, the corresponding value of the discrete subtype is assigned to the loop parameter. These values are assigned in increasing order unless the reserved word **reverse** is present, in which case the values are assigned in decreasing order.

5.5.1 User-defined Iterator Types

Insert after paragraph 11: [8652/0135]

An *iterable container type* is an indexable container type with specified *Default_Iterator* and *Iterator_Element* aspects. A *reversible iterable container type* is an iterable container type with the default iterator type being a reversible iterator type. An *iterable container object* is an object of an iterable container type. A *reversible iterable container object* is an object of a reversible iterable container type.

the new paragraph:

The *Default_Iterator* and *Iterator_Element* aspects are nonoverridable (see 13.1.1).

5.5.2 Generalized Loop Iteration

Replace paragraph 5: [8652/0145]

The type of the *subtype_indication*, if any, of an array component iterator shall cover the component type of the type of the *iterable_name*. The type of the *subtype_indication*, if any, of a container element iterator shall cover the default element type for the type of the *iterable_name*.

by:

The subtype defined by the *subtype_indication*, if any, of an array component iterator shall statically match the component subtype of the type of the *iterable_name*. The subtype defined by the *subtype_indication*, if any, of a container element iterator shall statically match the default element subtype for the type of the *iterable_name*.

Insert after paragraph 6: [8652/0146; 8652/0147]

In a container element iterator whose *iterable_name* has type *T*, if the *iterable_name* denotes a constant or the *Variable_Indexing* aspect is not specified for *T*, then the *Constant_Indexing* aspect shall be specified for *T*.

the new paragraphs:

The *iterator_name* or *iterable_name* of an *iterator_specification* shall not denote a subcomponent that depends on discriminants of an object whose nominal subtype is unconstrained, unless the object is known to be constrained.

A container element iterator is illegal if the call of the default iterator function that creates the loop iterator (see below) is illegal.

A generalized iterator is illegal if the iteration cursor subtype of the *iterator_name* is a limited type at the point of the generalized iterator. A container element iterator is illegal if the default cursor subtype of the type of the *iterable_name* is a limited type at the point of the container element iterator.

Insert after paragraph 13: [8652/0147]

For a forward container element iterator, the operation *First* of the iterator type is called on the loop iterator, to produce the initial value for the loop cursor. If the result of calling *Has_Element* on the initial value is *False*, then the execution of the *loop_statement* is complete. Otherwise, the *sequence_of_statements* is executed with the loop parameter denoting an indexing (see 4.1.6) into the iterable container object for the loop, with the only parameter to the indexing being the current value of the loop cursor; then the *Next* operation of the iterator type is called with the loop iterator and the loop cursor to produce the next value to be assigned to the loop cursor. This repeats until the result of calling *Has_Element* on the loop cursor is *False*, or until the loop is left as a consequence of a transfer of control. For a reverse container element iterator, the operations *Last* and *Previous* are called rather than *First* and *Next*. If the loop parameter is a constant (see above), then the indexing uses the default constant indexing function for the type of the iterable container object for the loop; otherwise it uses the default variable indexing function.

the new paragraph:

Any exception propagated by the execution of a generalized iterator or container element iterator is propagated by the immediately enclosing loop statement.