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*.

ICS 35.060 **Ref. No. ISO/IEC 8652:2012/Cor.1:2016(E)**

Published in Switzerland

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 *T*'Class). 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 *T*'Class).

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 <> 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 <> 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 <>, 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 <>, 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 satifies 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.

Section 6: Subprograms

6.1 Subprogram Declarations

Replace paragraph 39: [8652/0118]

```
function Min_Cell(X : Link) return Cell;                  -- see 3.10.1
function Next_Frame(K : Positive) return Frame;           -- see 3.10
function Dot_Product(Left, Right : Vector) return Real;   -- see 3.6
```

by:

```
function Min_Cell(X : Link) return Cell;                  -- see 3.10.1
function Next_Frame(K : Positive) return Frame;           -- see 3.10
function Dot_Product(Left, Right : Vector) return Real;   -- see 3.6
function Find(B : aliased in out Barrel; Key : String) return Real;
                                                          -- see 4.1.5
```

6.1.1 Preconditions and Postconditions

Replace paragraph 1: [8652/0148]

For a subprogram or entry, the following language-defined aspects may be specified with an aspect_specification (see 13.1.1):

by:

For a noninstance subprogram, a generic subprogram, or an entry, the following language-defined aspects may be specified with an aspect_specification (see 13.1.1):

Replace paragraph 7: [8652/0149; 8652/0125]

Within the expression for a Pre'Class or Post'Class aspect for a primitive subprogram of a tagged type *T*, a name that denotes a formal parameter of type *T* is interpreted as having type *T*'Class. Similarly, a name that denotes a formal access parameter of type access-to-*T* is interpreted as having type access-to-*T*'Class. This ensures that the expression is well-defined for a primitive subprogram of a type descended from *T*.

by:

Within the expression for a Pre'Class or Post'Class aspect for a primitive subprogram *S* of a tagged type *T*, a name that denotes a formal parameter (or *S*'Result) of type *T* is interpreted as though it had a (notional) type *NT* that is a formal derived type whose ancestor type is *T*, with directly visible primitive operations. Similarly, a name that denotes a formal access parameter (or *S*'Result) of type access-to-*T* is interpreted as having type access-to-*NT*. The result of this interpretation is that the only operations that can be applied to such names are those defined for such a formal derived type.

Insert after paragraph 17: [8652/0150]

If a renaming of a subprogram or entry *S1* overrides an inherited subprogram *S2*, then the overriding is illegal unless each class-wide precondition expression that applies to *S1* fully conforms to some class-wide precondition expression that applies to *S2* and each class-wide precondition expression that applies to *S2* fully conforms to some class-wide precondition expression that applies to *S1*.

the new paragraphs:

Pre'Class shall not be specified for an overriding primitive subprogram of a tagged type *T* unless the Pre'Class aspect is specified for the corresponding primitive subprogram of some ancestor of *T*.

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

Replace paragraph 18: [8652/0149; 8652/0150]

If a Pre'Class or Post'Class aspect is specified for a primitive subprogram of a tagged type *T*, then the associated expression also applies to the corresponding primitive subprogram of each descendant of *T*.

by:

If a Pre'Class or Post'Class aspect is specified for a primitive subprogram *S* of a tagged type *T*, or such an aspect defaults to True, then a corresponding expression also applies to the corresponding primitive subprogram *S* of each descendant of *T*. The *corresponding expression* is constructed from the associated expression as follows:

- References to formal parameters of *S* (or to *S* itself) are replaced with references to the corresponding formal parameters of the corresponding inherited or overriding subprogram *S* (or to the corresponding subprogram *S* itself).

The primitive subprogram *S* is illegal if it is not abstract and the corresponding expression for a Pre'Class or Post'Class aspect would be illegal.

Insert after paragraph 22: [8652/0134]

- a *dependent*_expression of a case_expression;

the new paragraph:

- the predicate of a quantified_expression;

Replace paragraph 26: [8652/0134; 8652/0125]

X'Old

For each X'Old in a postcondition expression that is enabled, a constant is implicitly declared at the beginning of the subprogram or entry. The constant is of the type of X and is initialized to the result of evaluating X (as an expression) at the point of the constant declaration. The value of X'Old in the postcondition expression is the value of this constant; the type of X'Old is the type of X. These implicit constant declarations occur in an arbitrary order.

by:

X'Old

Each X'Old in a postcondition expression that is enabled denotes a constant that is implicitly declared at the beginning of the subprogram body, entry body, or accept statement.

The implicitly declared entity denoted by each occurrence of X'Old is declared as follows:

- If X is of an anonymous access type defined by an access_definition *A* then

```
X'Old : constant A := X;
```

- If X is of a specific tagged type *T* then

```
anonymous : constant T'Class := T'Class (X);
X'Old : T renames T(anonymous);
```

where the name X'Old denotes the object renaming.

- Otherwise

```
X'Old : constant S := X;
```

where *S* is the nominal subtype of X. This includes the case where the type of *S* is an anonymous array type or a universal type.

The nominal subtype of X'Old is as implied by the above definitions. The expected type of the prefix of an Old attribute is that of the attribute. Similarly, if an Old attribute shall resolve to be of some type, then the prefix of the attribute shall resolve to be of that type.

Insert after paragraph 35: [8652/0134]

The precondition checks are performed in an arbitrary order, and if any of the class-wide precondition expressions evaluate to True, it is not specified whether the other class-wide precondition expressions are evaluated. The precondition checks and any check for elaboration of the subprogram body are performed in an arbitrary order. It is not specified whether in a call on a protected operation, the checks are performed before or after starting the protected action. For an entry call, the checks are performed prior to checking whether the entry is open.

the new paragraph:

For a call to a task entry, the postcondition check is performed before the end of the rendezvous; for a call to a protected operation, the postcondition check is performed before the end of the protected action of the call. The postcondition check for any call is performed before the finalization of any implicitly-declared constants associated (as described above) with Old attribute_references but after the finalization of any other entities whose accessibility level is that of the execution of the callable construct.

Replace paragraph 37: [8652/0149; 8652/0125]

For any subprogram or entry call (including dispatching calls), the checks that are performed to verify specific precondition expressions and specific and class-wide postcondition expressions are determined by those for the subprogram or entry actually invoked. Note that the class-wide postcondition expressions verified by the postcondition check that is part of a call on a primitive subprogram of type *T* includes all class-wide postcondition expressions originating in any progenitor of *T*, even if the primitive subprogram called is inherited from a type *T1* and some of the postcondition expressions do not apply to the corresponding primitive subprogram of *T1*.

by:

For any call to a subprogram or entry *S* (including dispatching calls), the checks that are performed to verify specific precondition expressions and specific and class-wide postcondition expressions are determined by those for the subprogram or entry actually invoked. Note that the class-wide postcondition expressions verified by the postcondition check that is part of a call on a primitive subprogram of type *T* includes all class-wide postcondition expressions originating in any progenitor of *T*, even if the primitive subprogram called is inherited from a type *T1* and some of the postcondition expressions do not apply to the corresponding primitive subprogram of *T1*. Any operations within a class-wide postcondition expression that were resolved as primitive operations of the (notional) formal derived type *NT*, are in the evaluation of the postcondition bound to the corresponding operations of the type identified by the controlling tag of the call on *S*. This applies to both dispatching and non-dispatching calls on *S*.

Replace paragraph 38: [8652/0149; 8652/0125]

The class-wide precondition check for a call to a subprogram or entry consists solely of checking the class-wide precondition expressions that apply to the denoted callable entity (not necessarily the one that is invoked).

by:

The class-wide precondition check for a call to a subprogram or entry *S* consists solely of checking the class-wide precondition expressions that apply to the denoted callable entity (not necessarily to the one that is invoked). Any operations within such an expression that were resolved as primitive operations of the (notional) formal derived type *NT* are in the evaluation of the precondition bound to the corresponding operations of the type identified by the controlling tag of the call on *S*. This applies to both dispatching and non-dispatching calls on *S*.

6.2 Formal Parameter Modes

Replace paragraph 10: [8652/0130]

A parameter of a by-reference type is passed by reference, as is an explicitly aliased parameter of any type. Each value of a by-reference type has an associated object. For a parenthesized expression, qualified_expression, or type_conversion, this object is the one associated with the operand. For a conditional_expression, this object is the one associated with the evaluated *dependent*_expression.

by:

A parameter of a by-reference type is passed by reference, as is an explicitly aliased parameter of any type. Each value of a by-reference type has an associated object. For a parenthesized expression, qualified_expression, or view conversion, this object is the one associated with the operand. For a value conversion, the associated object is the anonymous result object if such an object is created (see 4.6); otherwise it is the associated object of the operand. For a conditional_expression, this object is the one associated with the evaluated *dependent*_expression.

Replace paragraph 13: [8652/0118]

NOTES

6 A formal parameter of mode **in** is a constant view (see 3.3); it cannot be updated within the subprogram_body.

by:

NOTES

6 The mode of a formal parameter describes the direction of information transfer to or from the subprogram_body (see 6.1).

7 A formal parameter of mode **in** is a constant view (see 3.3); it cannot be updated within the subprogram_body.

8 A formal parameter of mode **out** might be uninitialized at the start of the subprogram_body (see 6.4.1).

6.3.1 Conformance Rules

Replace paragraph 10.1: [8652/0151]

- any prefixed view of a subprogram (see 4.1.3).

by:

- any prefixed view of a subprogram (see 4.1.3) without synchronization kind (see 9.5) By_Entry or By_Protected_Procedure.

Replace paragraph 12: [8652/0151; 8652/0125]

- The default calling convention is *protected* for a protected subprogram, and for an access-to-subprogram type with the reserved word **protected** in its definition.

by:

- The default calling convention is *protected* for a protected subprogram, for a prefixed view of a subprogram with a synchronization kind of By_Protected_Procedure, and for an access-to-subprogram type with the reserved word **protected** in its definition.

Replace paragraph 13: [8652/0151; 8652/0125]

- The default calling convention is *entry* for an entry.

by:

- The default calling convention is *entry* for an entry and for a prefixed view of a subprogram with a synchronization kind of By_Entry.

Insert after paragraph 20: [8652/0152]

- each constituent construct of one corresponds to an instance of the same syntactic category in the other, except that an expanded name may correspond to a direct_name (or character_literal) or to a different expanded name in the other; and

the new paragraph:

- corresponding defining_identifiers occurring within the two expressions are the same; and

Replace paragraph 21: [8652/0152]

- each direct_name, character_literal, and selector_name that is not part of the prefix of an expanded name in one denotes the same declaration as the corresponding direct_name, character_literal, or selector_name in the other; and

by:

- each direct_name, character_literal, and selector_name that is not part of the prefix of an expanded name in one denotes the same declaration as the corresponding direct_name, character_literal, or selector_name in the other, or they denote corresponding declarations occurring within the two expressions; and

6.4.1 Parameter Associations

Insert after paragraph 5: [8652/0142]

If the mode is **in out** or **out**, the actual shall be a name that denotes a variable.

the new paragraph:

If the mode is **out**, the actual parameter is a view conversion, and the type of the formal parameter is an access type or a scalar type that has the Default_Value aspect specified, then

- there shall exist a type (other than a root numeric type) that is an ancestor of both the target type and the operand type; and
- in the case of a scalar type, the type of the operand of the conversion shall have the Default_Value aspect specified.

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

Insert after paragraph 6.2: [8652/0133]

- the subtype *F* shall be unconstrained, discriminated in its full view, and unconstrained in any partial view.

the new paragraph:

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

Replace paragraph 13.1: [8652/0142; 8652/0125]

- For a scalar type that has the Default_Value aspect specified, the formal parameter is initialized from the value of the actual, without checking that the value satisfies any constraint or any predicate;

by:

- For a scalar type that has the Default_Value aspect specified, the formal parameter is initialized from the value of the actual, without checking that the value satisfies any constraint or any predicate. Furthermore, if the actual parameter is a view conversion and either
  - there exists no type (other than a root numeric type) that is an ancestor of both the target type and the type of the operand of the conversion; or
  - the Default_Value aspect is unspecified for the type of the operand of the conversion

then Program_Error is raised;

6.5 Return Statements

Replace paragraph 8: [8652/0153]

If the result type of a function is a specific tagged type, the tag of the return object is that of the result type. If the result type is class-wide, the tag of the return object is that of the type of the subtype_indication if it is specific, or otherwise that of the value of the expression. A check is made that the master of the type identified by the tag of the result includes the elaboration of the master that elaborated the function body. If this check fails, Program_Error is raised.

by:

If the result type of a function is a specific tagged type, the tag of the return object is that of the result type. If the result type is class-wide, the tag of the return object is that of the value of the expression, unless the return object is defined by an extended_return_object_declaration with a subtype_indication that is specific, in which case it is that of the type of the subtype_indication. A check is made that the master of the type identified by the tag of the result includes the elaboration of the master that elaborated the function body. If this check fails, Program_Error is raised.

6.8 Expression Functions

Replace paragraph 2: [8652/0132]

```
expression_function_declaration ::=
   [overriding_indicator]
   function_specification is
      (expression)
      [aspect_specification];
```

by:

```
expression_function_declaration ::=
   [overriding_indicator]
   function_specification is
      (expression)
```

```
        [aspect_specification];
| [overriding_indicator]
  function_specification is
     aggregate
     [aspect_specification];
```

Replace paragraph 3: [8652/0132]

The expected type for the expression of an expression_function_declaration is the result type (see 6.5) of the function.

by:

The expected type for the expression or aggregate of an expression_function_declaration is the result type (see 6.5) of the function.

Replace paragraph 5: [8652/0132]

If the result subtype has one or more unconstrained access discriminants, the accessibility level of the anonymous access type of each access discriminant, as determined by the expression of the expression function, shall not be statically deeper than that of the master that elaborated the expression_function_declaration.

by:

If the result subtype has one or more unconstrained access discriminants, the accessibility level of the anonymous access type of each access discriminant, as determined by the expression or aggregate of the expression_function_declaration, shall not be statically deeper than that of the master that elaborated the expression_function_declaration.

Replace paragraph 6: [8652/0132]

An expression_function_declaration declares an *expression function*. A completion is not allowed for an expression_function_declaration; however, an expression_function_declaration can complete a previous declaration.

by:

An expression_function_declaration declares an *expression function*. The *return expression* of an expression function is the expression or aggregate of the expression_function_declaration. A completion is not allowed for an expression_function_declaration; however, an expression_function_declaration can complete a previous declaration.

Replace paragraph 7: [8652/0132]

The execution of an expression function is invoked by a subprogram call. For the execution of a subprogram call on an expression function, the execution of the subprogram_body executes an implicit function body containing only a simple_return_statement whose expression is that of the expression function.

by:

The execution of an expression function is invoked by a subprogram call. For the execution of a subprogram call on an expression function, the execution of the subprogram_body executes an implicit function body containing only a simple_return_statement whose expression is the return expression of the expression function.

Section 7: Packages

7.3.1 Private Operations

Replace paragraph 5.2: [8652/0154]

It is possible for there to be places where a derived type is visibly a descendant of an ancestor type, but not a descendant of even a partial view of the ancestor type, because the parent of the derived type is not visibly a descendant of the ancestor. In this case, the derived type inherits no characteristics from that ancestor, but nevertheless is within the derivation class of the ancestor for the purposes of type conversion, the "covers" relationship, and matching against a formal derived type. In this case the derived type is considered to be a *descendant* of an incomplete view of the ancestor.

by:

Furthermore, it is possible for there to be places where a derived type is known to be derived indirectly from an ancestor type, but is not a descendant of even a partial view of the ancestor type, because the parent of the derived type is not visibly a descendant of the ancestor. In this case, the derived type inherits no characteristics from that ancestor, but nevertheless is within the derivation class of the ancestor for the purposes of type conversion, the "covers" relationship, and matching against a formal derived type. In this case the derived type is effectively a *descendant* of an incomplete view of the ancestor.

7.3.2 Type Invariants

Replace paragraph 1: [8652/0155]

For a private type or private extension, the following language-defined aspects may be specified with an aspect_specification (see 13.1.1):

by:

For a private type, private extension, or interface, the following language-defined aspects may be specified with an aspect_specification (see 13.1.1):

Replace paragraph 3: [8652/0155; 8652/0156]

Type_Invariant'Class
: This aspect shall be specified by an expression, called an *invariant expression*. Type_Invariant'Class may be specified on a private_type_declaration or a private_extension_declaration.

by:

Type_Invariant'Class
: This aspect shall be specified by an expression, called an *invariant expression*. Type_Invariant'Class may be specified on a private_type_declaration, a private_extension_declaration, or a full_type_declaration for an interface type. Type_Invariant'Class determines a *class-wide type invariant* for a tagged type.

Replace paragraph 5: [8652/0156; 8652/0125]

Within an invariant expression, the identifier of the first subtype of the associated type denotes the current instance of the type. Within an invariant expression associated with type *T*, the type of the current instance is *T* for the Type_Invariant aspect and *T*'Class for the Type_Invariant'Class aspect.

by:

Within an invariant expression, the identifier of the first subtype of the associated type denotes the current instance of the type. Within an invariant expression for the Type_Invariant aspect of a type *T*, the type of this current instance is *T*. Within an invariant expression for the Type_Invariant'Class aspect of a type *T*, the type of this current instance is interpreted as though it had a (notional) type *NT* that is a visible formal derived type whose ancestor type is *T*. The effect of this interpretation is that the only operations that can be applied to this current instance are those defined for such a formal derived type.

Insert after paragraph 6: [8652/0157]

The Type_Invariant'Class aspect shall not be specified for an untagged type. The Type_Invariant aspect shall not be specified for an abstract type.

the new paragraph:

If a type extension occurs at a point where a private operation of some ancestor is visible and inherited, and a Type_Invariant'Class expression applies to that ancestor, then the inherited operation shall be abstract or shall be overridden.

Replace paragraph 9: [8652/0156]

If one or more invariant expressions apply to a type *T*, then an invariant check is performed at the following places, on the specified object(s):

by:

If one or more invariant expressions apply to a nonabstract type *T*, then an invariant check is performed at the following places, on the specified object(s):

Replace paragraph 10: [8652/0158; 8652/0159]

- After successful default initialization of an object of type *T*, the check is performed on the new object;

by:

- After successful initialization of an object of type *T* by default (see 3.3.1), the check is performed on the new object unless the partial view of *T* has unknown discriminants;
- After successful explicit initialization of the completion of a deferred constant with a part of type *T*, if the completion is inside the immediate scope of the full view of *T*, and the deferred constant is visible outside the immediate scope of *T*, the check is performed on the part(s) of type *T*;

Replace paragraph 15: [8652/0160]

- After a successful call on the Read or Input stream attribute of the type *T*, the check is performed on the object initialized by the stream attribute;

by:

- After a successful call on the Read or Input stream-oriented attribute of the type *T*, the check is performed on the object initialized by the attribute;

Replace paragraph 17: [8652/0157]

- is declared within the immediate scope of type *T* (or by an instance of a generic unit, and the generic is declared within the immediate scope of type *T*), and

by:

- is declared within the immediate scope of type *T* (or by an instance of a generic unit, and the generic is declared within the immediate scope of type *T*),

Delete paragraph 18: [8652/0157]

- is visible outside the immediate scope of type *T* or overrides an operation that is visible outside the immediate scope of *T*, and

Replace paragraph 19: [8652/0157; 8652/0161; 8652/0162]

- has a result with a part of type *T*, or one or more parameters with a part of type *T*, or an access to variable parameter whose designated type has a part of type *T*.

by:

- and either:
  - has a result with a part of type *T*, or
  - has one or more **out** or **in out** parameters with a part of type *T*, or
  - has an access-to-object parameter or result whose designated type has a part of type *T*, or
  - is a procedure or entry that has an **in** parameter with a part of type *T*,
- and either:

- *T* is a private type or a private extension and the subprogram or entry is visible outside the immediate scope of type *T* or overrides an inherited operation that is visible outside the immediate scope of *T*, or
- *T* is a record extension, and the subprogram or entry is a primitive operation visible outside the immediate scope of type *T* or overrides an inherited operation that is visible outside the immediate scope of *T*.

Insert after paragraph 20: [8652/0157]

The check is performed on each such part of type *T*.

the new paragraph:

- For a view conversion to a class-wide type occurring within the immediate scope of *T*, from a specific type that is a descendant of *T* (including *T* itself), a check is performed on the part of the object that is of type *T*.

Replace paragraph 21: [8652/0126; 8652/0125]

If performing checks is required by the Invariant or Invariant'Class assertion policies (see 11.4.2) in effect at the point of corresponding aspect specification applicable to a given type, then the respective invariant expression is considered *enabled*.

by:

If performing checks is required by the Type_Invariant or Type_Invariant'Class assertion policies (see 11.4.2) in effect at the point of the corresponding aspect specification applicable to a given type, then the respective invariant expression is considered *enabled*.

Insert after paragraph 22: [8652/0156; 8652/0125]

The invariant check consists of the evaluation of each enabled invariant expression that applies to *T*, on each of the objects specified above. If any of these evaluate to False, Assertions.Assertion_Error is raised at the point of the object initialization, conversion, or call. If a given call requires more than one evaluation of an invariant expression, either for multiple objects of a single type or for multiple types with invariants, the evaluations are performed in an arbitrary order, and if one of them evaluates to False, it is not specified whether the others are evaluated. Any invariant check is performed prior to copying back any by-copy **in out** or **out** parameters. Invariant checks, any postcondition check, and any constraint or predicate checks associated with **in out** or **out** parameters are performed in an arbitrary order.

the new paragraph:

For an invariant check on a value of type *T1* based on a class-wide invariant expression inherited from an ancestor type *T*, any operations within the invariant expression that were resolved as primitive operations of the (notional) formal derived type *NT* are bound to the corresponding operations of type *T1* in the evaluation of the invariant expression for the check on *T1*.

7.5 Limited Types

Replace paragraph 2.9: [8652/0132]

- the expression of an expression_function_declaration (see 6.8)

by:

- the return expression of an expression function (see 6.8)

Section 8: Visibility Rules

8.1 Declarative Region

Insert after paragraph 2: [8652/0163]

- any declaration, other than that of an enumeration type, that is not a completion of a previous declaration;

the new paragraph:

- an access_definition;

8.2 Scope of Declarations

Insert after paragraph 11: [8652/0164]

The immediate scope of a declaration is also the immediate scope of the entity or view declared by the declaration. Similarly, the scope of a declaration is also the scope of the entity or view declared by the declaration.

the new paragraph:

The immediate scope of a pragma that is not used as a configuration pragma is defined to be the region extending from immediately after the pragma to the end of the declarative region immediately enclosing the pragma.

8.6 The Context of Overload Resolution

Replace paragraph 9: [8652/0165]

- The expression of a case_statement.

by:

- The *selecting*_expression of a case_statement or case_expression.

Insert after paragraph 17: [8652/0166]

- If a usage name appears within the declarative region of a type_declaration and denotes that same type_declaration, then it denotes the *current instance* of the type (rather than the type itself); the current instance of a type is the object or value of the type that is associated with the execution that evaluates the usage name. Similarly, if a usage name appears within the declarative region of a subtype_declaration and denotes that same subtype_declaration, then it denotes the current instance of the subtype. These rules do not apply if the usage name appears within the subtype_mark of an access_definition for an access-to-object type, or within the subtype of a parameter or result of an access-to-subprogram type.

the new paragraph:

Within an aspect_specification for a type or subtype, the current instance represents a value of the type; it is not an object. The nominal subtype of this value is given by the subtype itself (the first subtype in the case of a type_declaration), prior to applying any predicate specified directly on the type or subtype. If the type or subtype is by-reference, the associated object with the value is the object associated (see 6.2) with the execution of the usage name.

Replace paragraph 27.1: [8652/0122]

Other than for the simple_expression of a membership test, if the expected type for a name or expression is not the same as the actual type of the name or expression, the actual type shall be convertible to the expected type (see 4.6); further, if the expected type is a named access-to-object type with designated type *D1* and the actual type is an anonymous access-to-object type with designated type *D2*, then *D1* shall cover *D2*, and the name or expression shall denote a view with an accessibility level for which the statically deeper relationship applies; in particular it shall not denote an access parameter nor a stand-alone access object.

by:

Other than for the *tested*_simple_expression of a membership test, if the expected type for a name or expression is not the same as the actual type of the name or expression, the actual type shall be convertible to the expected type (see 4.6); further, if the expected type is a named access-to-object type with designated type *D1* and the actual type is an anonymous access-to-object type with designated type *D2*, then *D1* shall cover *D2*, and the name or expression shall denote a view

with an accessibility level for which the statically deeper relationship applies; in particular it shall not denote an access parameter nor a stand-alone access object.

Section 9: Tasks and Synchronization

9.3 Task Dependence - Termination of Tasks

Replace paragraph 2: [8652/0131]

- If the task is created by the evaluation of an allocator for a given access type, it depends on each master that includes the elaboration of the declaration of the ultimate ancestor of the given access type.

by:

- If the task is created by the evaluation of an allocator for a given named access type, it depends on each master that includes the elaboration of the declaration of the ultimate ancestor of the given access type.

9.4 Protected Units and Protected Objects

Replace paragraph 8: [8652/0167]

```
protected_operation_item ::= subprogram_declaration
                           | subprogram_body
                           | entry_body
                           | aspect_clause
```

by:

```
protected_operation_item ::= subprogram_declaration
                           | subprogram_body
                           | null_procedure_declaration
                           | expression_function_declaration
                           | entry_body
                           | aspect_clause
```

9.5.1 Protected Subprograms and Protected Actions

Insert after paragraph 2: [8652/0168]

Within the body of a protected function (or a function declared immediately within a protected_body), the current instance of the enclosing protected unit is defined to be a constant (that is, its subcomponents may be read but not updated). Within the body of a protected procedure (or a procedure declared immediately within a protected_body), and within an entry_body, the current instance is defined to be a variable (updating is permitted).

the new paragraphs:

For a type declared by a protected_type_declaration or for the anonymous type of an object declared by a single_protected_declaration, the following language-defined type-related representation aspect may be specified:

Exclusive_Functions

The type of aspect Exclusive_Functions is Boolean. If not specified (including by inheritance), the aspect is False.

A value of True for this aspect indicates that protected functions behave in the same way as protected procedures with respect to mutual exclusion and queue servicing (see below).

A protected procedure or entry is an *exclusive* protected operation. A protected function of a protected type *P* is an exclusive protected operation if the Exclusive_Functions aspect of *P* is True.

Replace paragraph 4: [8652/0168]

A new protected action is not started on a protected object while another protected action on the same protected object is underway, unless both actions are the result of a call on a protected function. This rule is expressible in terms of the execution resource associated with the protected object:

by:

A new protected action is not started on a protected object while another protected action on the same protected object is underway, unless both actions are the result of a call on a nonexclusive protected function. This rule is expressible in terms of the execution resource associated with the protected object:

Replace paragraph 5: [8652/0168]

- *Starting* a protected action on a protected object corresponds to *acquiring* the execution resource associated with the protected object, either either for concurrent read-only access if the protected action is for a call on a protected function, or for exclusive read-write access otherwise;

by:

- *Starting* a protected action on a protected object corresponds to *acquiring* the execution resource associated with the protected object, either for exclusive read-write access if the protected action is for a call on an exclusive protected operation, or for concurrent read-only access otherwise;

Replace paragraph 7: [8652/0168]

After performing an operation on a protected object other than a call on a protected function, but prior to completing the associated protected action, the entry queues (if any) of the protected object are serviced (see 9.5.3).

by:

After performing an exclusive protected operation on a protected object, but prior to completing the associated protected action, the entry queues (if any) of the protected object are serviced (see 9.5.3).

9.5.3 Entry Calls

Replace paragraph 15: [8652/0168]

- If after performing, as part of a protected action on the associated protected object, an operation on the object other than a call on a protected function, the entry is checked and found to be open.

by:

- If after performing, as part of a protected action on the associated protected object, an exclusive protected operation on the object, the entry is checked and found to be open.

Replace paragraph 23: [8652/0168]

When the entry of a protected object is checked to see whether it is open, the implementation need not reevaluate the condition of the corresponding entry_barrier if no variable or attribute referenced by the condition (directly or indirectly) has been altered by the execution (or cancellation) of a protected procedure or entry call on the object since the condition was last evaluated.

by:

When the entry of a protected object is checked to see whether it is open, the implementation need not reevaluate the condition of the corresponding entry_barrier if no variable or attribute referenced by the condition (directly or indirectly) has been altered by the execution (or cancellation) of a call to an exclusive protected operation of the object since the condition was last evaluated.

9.5.4 Requeue Statements

Insert after paragraph 5: [8652/0169]

If the requeue target has parameters, then its (prefixed) profile shall be subtype conformant with the profile of the innermost enclosing callable construct.

the new paragraphs:

Given a requeue_statement where the innermost enclosing callable construct is for an entry *E1*, for every specific or class-wide postcondition expression *P1* that applies to *E1*, there shall exist a postcondition expression *P2* that applies to the requeue target *E2* such that

- *P1* is fully conformant with the expression produced by replacing each reference in *P2* to a formal parameter of *E2* with a reference to the corresponding formal paramter of *E1*; and
- if *P1* is enabled, then *P2* is also enabled.

The requeue target shall not have an applicable specific or class-wide postcondition which includes an Old attribute_reference.

If the requeue target is declared immediately within the task_definition of a named task type or the protected_definition of a named protected type, and if the requeue statement occurs within the body of that type, and if the requeue is an external requeue, then the requeue target shall not have a specific or class-wide postcondition which includes a name denoting either the current instance of that type or any entity declared within the declaration of that type.

Replace paragraph 7: [8652/0169]

The execution of a requeue_statement proceeds by first evaluating the *procedure_or_entry_*name, including the prefix identifying the target task or protected object and the expression identifying the entry within an entry family, if any. The entry_body or accept_statement enclosing the requeue_statement is then completed, finalized, and left (see 7.6.1).

by:

The execution of a requeue_statement proceeds by first evaluating the *procedure_or_entry_*name, including the prefix identifying the target task or protected object and the expression identifying the entry within an entry family, if any. Precondition checks are then performed as for a call to the requeue target entry or subprogram. The entry_body or accept_statement enclosing the requeue_statement is then completed, finalized, and left (see 7.6.1).

Replace paragraph 12: [8652/0169]

If the requeue target named in the requeue_statement has formal parameters, then during the execution of the accept_statement or entry_body corresponding to the new entry, the formal parameters denote the same objects as did the corresponding formal parameters of the callable construct completed by the requeue. In any case, no parameters are specified in a requeue_statement; any parameter passing is implicit.

by:

If the requeue target named in the requeue_statement has formal parameters, then during the execution of the accept_statement or entry_body corresponding to the new entry and during the checking of any preconditions of the new entry, the formal parameters denote the same objects as did the corresponding formal parameters of the callable construct completed by the requeue. In any case, no parameters are specified in a requeue_statement; any parameter passing is implicit.

9.7.4 Asynchronous Transfer of Control

Insert after paragraph 13: [8652/0170]

```
select
   delay 5.0;
   Put_Line("Calculation does not converge");
then abort
   -- This calculation should finish in 5.0 seconds;
   --  if not, it is assumed to diverge.
   Horribly_Complicated_Recursive_Function(X, Y);
end select;
```

the new paragraph:

Note that these examples presume that there are abort completion points within the execution of the abortable_part.

Section 10: Program Structure and Compilation Issues

10.2.1 Elaboration Control

Insert after paragraph 17: [8652/0171]

A pragma Pure is used to specify that a library unit is *declared pure*, namely that the Pure aspect of the library unit is True; all compilation units of the library unit are declared pure. In addition, the limited view of any library package is declared pure. The declaration and body of a declared pure library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be pure. All compilation units of a declared pure library unit shall depend semantically only on declared pure library_items. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. Furthermore, the full view of any partial view declared in the visible part of a declared pure library unit that has any available stream attributes shall support external streaming (see 13.13.2).

the new paragraph:

Erroneous Execution

Execution is erroneous if some operation (other than the initialization or finalization of the object) modifies the value of a constant object declared at library-level in a pure package.

Section 11: Exceptions

11.2 Exception Handlers

Insert before paragraph 6: [8652/0139]

A choice with an *exception*_name *covers* the named exception. A choice with **others** covers all exceptions not named by previous choices of the same handled_sequence_of_statements. Two choices in different exception_handlers of the same handled_sequence_of_statements shall not cover the same exception.

the new paragraph:

An *exception*_name of an exception_choice shall denote an exception.

11.3 Raise Statements and Raise Expressions

Insert after paragraph 2: [8652/0139; 8652/0124]

raise_statement ::= **raise**; |
raise *exception*_name [**with** *string*_expression];

the new paragraphs:

raise_expression ::= **raise** *exception*_name [**with** *string*_simple_expression]

If a raise_expression appears within the expression of one of the following contexts, the raise_expression shall appear within a pair of parentheses within the expression:

- object_declaration;
- modular_type_definition;
- floating_point_definition;
- ordinary_fixed_point_definition;
- decimal_fixed_point_definition;
- default_expression;
- ancestor_part.

Replace paragraph 3: [8652/0139; 8652/0125]

The name, if any, in a raise_statement shall denote an exception. A raise_statement with no *exception*_name (that is, a *re-raise statement*) shall be within a handler, but not within a body enclosed by that handler.

by:

The *exception*_name, if any, of a raise_statement or raise_expression shall denote an exception. A raise_statement with no *exception*_name (that is, a *re-raise statement*) shall be within a handler, but not within a body enclosed by that handler.

Replace paragraph 3.1: [8652/0139; 8652/0124; 8652/0125]

The expression, if any, in a raise_statement, is expected to be of type String.

by:

The *string*_expression or *string*_simple_expression, if any, of a raise_statement or raise_expression is expected to be of type String.

The expected type for a raise_expression shall be any single type.

Replace paragraph 4: [8652/0139; 8652/0172; 8652/0124; 8652/0000]

To *raise an exception* is to raise a new occurrence of that exception, as explained in 11.4. For the execution of a raise_statement with an *exception*_name, the named exception is raised. If a *string*_expression is present, the expression is evaluated and its value is associated with the exception occurrence. For the execution of a re-raise statement, the exception occurrence that caused transfer of control to the innermost enclosing handler is raised again.

by:

To *raise an exception* is to raise a new occurrence of that exception, as explained in 11.4. For the execution of a raise_statement with an *exception*_name, the named exception is raised. Similarly, for the evaluation of a raise_expression, the named exception is raised. In both of these cases, if a *string*_expression or *string*_simple_expression is present, the expression is evaluated and its value is associated with the exception occurrence. For the execution of a re-raise statement, the exception occurrence that caused transfer of control to the innermost enclosing handler is raised again.

NOTES

1 If the evaluation of a *string*_expression or *string*_simple_expression raises an exception, that exception is propagated instead of the one denoted by the *exception*_name of the raise_statement or raise_expression.

11.4.1 The Package Exceptions

Replace paragraph 10.1: [8652/0139; 8652/0124]

Exception_Message returns the message associated with the given Exception_Occurrence. For an occurrence raised by a call to Raise_Exception, the message is the Message parameter passed to Raise_Exception. For the occurrence raised by a raise_statement with an *exception*_name and a *string*_expression, the message is the *string*_expression. For the occurrence raised by a raise_statement with an *exception*_name but without a *string*_expression, the message is a string giving implementation-defined information about the exception occurrence. For an occurrence originally raised in some other manner (including by the failure of a language-defined check), the message is an unspecified string. In all cases, Exception_Message returns a string with lower bound 1.

by:

Exception_Message returns the message associated with the given Exception_Occurrence. For an occurrence raised by a call to Raise_Exception, the message is the Message parameter passed to Raise_Exception. For the occurrence raised by a raise_statement or raise_expression with an *exception*_name and a *string*_expression or *string*_simple_expression, the message is the *string*_expression or *string*_simple_expression. For the occurrence raised by a raise_statement or raise_expression with an *exception*_name but without a *string*_expression or *string*_simple_expression, the message is a string giving implementation-defined information about the exception occurrence. For an occurrence originally raised in some other manner (including by the failure of a language-defined check), the message is an unspecified string. In all cases, Exception_Message returns a string with lower bound 1.

Section 12: Generic Units

12.5.1 Formal Private and Derived Types

Replace paragraph 5.1: [8652/0173]

The actual type for a formal derived type shall be a descendant of the ancestor type and every progenitor of the formal type. If the formal type is nonlimited, the actual type shall be nonlimited. If the reserved word **synchronized** appears in the declaration of the formal derived type, the actual type shall be a synchronized tagged type.

by:

The actual type for a formal derived type shall be a descendant of the ancestor type and every progenitor of the formal type. If the formal type is nonlimited, the actual type shall be nonlimited. The actual type for a formal derived type shall be tagged if and only if the formal derived type is a private extension. If the reserved word **synchronized** appears in the declaration of the formal derived type, the actual type shall be a synchronized tagged type.

Insert after paragraph 15: [8652/0133]

For a generic formal type with an unknown_discriminant_part, the actual may, but need not, have discriminants, and may be definite or indefinite.

the new paragraph:

When enforcing Legality Rules, for the purposes of determining within a generic body whether a type 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.

Section 13: Representation Issues

13.1 Operational and Representation Aspects

Replace paragraph 9: [8652/0174]

A representation item that directly specifies an aspect of a subtype or type shall appear after the type is completely defined (see 3.11.1), and before the subtype or type is frozen (see 13.14). If a representation item or aspect_specification is given that directly specifies an aspect of an entity, then it is illegal to give another representation item or aspect_specification that directly specifies the same aspect of the entity.

by:

A representation item that directly specifies an aspect of a subtype or type shall appear after the type is completely defined (see 3.11.1), and before the subtype or type is frozen (see 13.14).

Replace paragraph 9.1: [8652/0174]

An operational item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.14). If an operational item or aspect_specification is given that directly specifies an aspect of an entity, then it is illegal to give another operational item or aspect_specification that directly specifies the same aspect of the entity.

by:

An operational item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.14).

If a representation item, operational item, or aspect_specification is given that directly specifies an aspect of an entity, then it is illegal to give another representation item, operational item, or aspect_specification that directly specifies the same aspect of the entity.

Replace paragraph 10: [8652/0175]

For an untagged derived type, it is illegal to specify a type-related representation aspect if the parent type is a by-reference type, or has any user-defined primitive subprograms.

by:

For an untagged derived type, it is illegal to specify a type-related representation aspect if the parent type is a by-reference type, or has any user-defined primitive subprograms. Similarly, it is illegal to specify a nonconfirming type-related representation aspect for an untagged by-reference type after one or more types have been derived from it.

13.1.1 Aspect Specifications

Replace paragraph 18: [8652/0176; 8652/0135]

A language-defined aspect shall not be specified in an aspect_specification given on a subprogram_body or subprogram_body_stub that is a completion of another declaration.

by:

A language-defined aspect shall not be specified in an aspect_specification given on a completion of a subprogram or generic subprogram.

If an aspect of a derived type is inherited from an ancestor type and has the boolean value True, the inherited value shall not be overridden to have the value False for the derived type, unless otherwise specified in this International Standard.

Certain type-related aspects are defined to be *nonoverridable*; all such aspects are specified using an aspect_definition that is a name.

If a nonoverridable aspect is directly specified for a type *T*, then any explicit specification of that aspect for any other descendant of *T* shall be confirming; that is, the specified name shall *match* the inherited aspect, meaning that the specified name shall denote the same declarations as would the inherited name.

If a full type has a partial view, and a given nonoverridable aspect is allowed for both the full view and the partial view, then the given aspect for the partial view and the full view shall be the same: the aspect shall be directly specified only on the partial view; if the full type inherits the aspect, then a matching definition shall be specified (directly or by inheritance) for the partial view.

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

The Default_Iterator, Iterator_Element, Implicit_Dereference, Constant_Indexing, and Variable_Indexing aspects are nonoverridable.

Replace paragraph 28: [8652/0177]

If the aspect_mark includes 'Class, then:

by:

If the aspect_mark includes 'Class (a *class-wide aspect*), then, unless specified otherwise for a particular class-wide aspect:

Replace paragraph 32: [8652/0178]

Any aspect specified by a representation pragma or library unit pragma that has a local_name as its single argument may be specified by an aspect_specification, with the entity being the local_name. The aspect_definition is expected to be of type Boolean. The expression shall be static.

by:

Any aspect specified by a representation pragma or library unit pragma that has a local_name as its single argument may be specified by an aspect_specification, with the entity being the local_name. The aspect_definition is expected to be of type Boolean. The expression shall be static. Notwithstanding what this International Standard says elsewhere, the expression of an aspect that can be specified by a library unit pragma is resolved and evaluated at the point where it occurs in the aspect_specification, rather than the first freezing point of the associated package.

Delete paragraph 34: [8652/0135]

If an aspect of a derived type is inherited from an ancestor type and has the boolean value True, the inherited value shall not be overridden to have the value False for the derived type, unless otherwise specified in this International Standard.

13.2 Packed Types

Delete paragraph 6.1: [8652/0179]

If a packed type has a component that is not of a by-reference type and has no aliased part, then such a component need not be aligned according to the Alignment of its subtype; in particular it need not be allocated on a storage element boundary.

Insert after paragraph 7: [8652/0179]

The recommended level of support for pragma Pack is:

the new paragraph:

- Any component of a packed type that is of a by-reference type, that is specified as independently addressable, or that contains an aliased part, shall be aligned according to the alignment of its subtype.

Replace paragraph 8: [8652/0179]

- For a packed record type, the components should be packed as tightly as possible subject to, the Sizes of the component subtypes, and subject to any record_representation_clause that applies to the type; the implementation may, but need not, reorder components or cross aligned word boundaries to improve the packing. A component whose Size is greater than the word size may be allocated an integral number of words.

by:

- For a packed record type, the components should be packed as tightly as possible subject to the above alignment requirements, the Sizes of the component subtypes, and any record_representation_clause that applies to the type; the implementation may, but need not, reorder components or cross aligned word boundaries to improve the packing. A component whose Size is greater than the word size may be allocated an integral number of words.

Replace paragraph 9: [8652/0179]

- For a packed array type, if the Size of the component subtype is less than or equal to the word size, Component_Size should be less than or equal to the Size of the component subtype, rounded up to the nearest factor of the word size.

by:

- For a packed array type, if the Size of the component subtype is less than or equal to the word size, Component_Size should be less than or equal to the Size of the component subtype, rounded up to the nearest factor of the word size, unless this would violate the above alignment requirements.

13.3 Representation Attributes

Replace paragraph 73.4: [8652/0180]

The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved and returns True if the representation of the object denoted by the actual parameter occupies exactly the same bits as the representation of the object denoted by X; otherwise, it returns False.

by:

The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved. It returns True if the representation of the object denoted by the actual parameter occupies exactly the same bits as the representation of the object denoted by X and the objects occupy at least one bit; otherwise, it returns False.

13.9.2 The Valid Attribute

Replace paragraph 3: [8652/0119]

X'Valid

Yields True if and only if the object denoted by X is normal, has a valid representation, and the predicate of the nominal subtype of X evaluates to True. The value of this attribute is of the predefined type Boolean.

by:

X'Valid

Yields True if and only if the object denoted by X is normal, has a valid representation, and then, if the preceding conditions hold, the value of X also satisfies the predicates of the nominal subtype of X. The value of this attribute is of the predefined type Boolean.

Replace paragraph 12: [8652/0119]

23 X'Valid is not considered to be a read of X; hence, it is not an error to check the validity of invalid data.

by:

23 Determining whether X is normal and has a valid representation as part of the evaluation of X'Valid is not considered to include an evaluation of X; hence, it is not an error to check the validity of an object that is invalid or abnormal. Determining whether X satisfies the predicates of its nominal subtype may include an evaluation of X, but only after it has been determined that X has a valid representation.

If X is volatile, the evaluation of X'Valid is considered a read of X.

13.11 Storage Management

Replace paragraph 18: [8652/0181]

If Storage_Size is specified for an access type, then the Storage_Size of this pool is at least that requested, and the storage for the pool is reclaimed when the master containing the declaration of the access type is left. If the implementation cannot satisfy the request, Storage_Error is raised at the point of the attribute_definition_clause. If neither Storage_Pool nor Storage_Size are specified, then the meaning of Storage_Size is implementation defined.

by:

If Storage_Size is specified for an access type *T*, an implementation-defined pool *P* is used for the type. The Storage_Size of *P* is at least that requested, and the storage for *P* is reclaimed when the master containing the declaration of the access type is left. If the implementation cannot satisfy the request, Storage_Error is raised at the freezing point of type *T*. The storage pool *P* is used only for allocators returning type *T* or other access types specified to use *T*'Storage_Pool. Storage_Error is raised by an allocator returning such a type if the storage space of *P* is exhausted (additional memory is not allocated).

If neither Storage_Pool nor Storage_Size are specified, then the meaning of Storage_Size is implementation defined.

13.11.2 Unchecked Storage Deallocation

Replace paragraph 10: [8652/0182]

After Free(X), the object designated by X, and any subcomponents (and coextensions) thereof, no longer exist; their storage can be reused for other purposes.

by:

After the finalization step of Free(X), the object designated by X, and any subcomponents (and coextensions) thereof, no longer exist; their storage can be reused for other purposes.

Insert after paragraph 15: [8652/0182]

In the first two cases, the storage for the discriminants (and for any enclosing object if it is designated by an access discriminant of the task) is not reclaimed prior to task termination.

the new paragraphs:

An access value that designates a nonexistent object is called a *dangling reference*.

If a dangling reference is dereferenced (implicitly or explicitly), execution is erroneous (see below). If there is no explicit or implicit dereference, then it is a bounded error to evaluate an expression whose result is a dangling reference. If the error is detected, either Constraint_Error or Program_Error is raised. Otherwise, execution proceeds normally, but with the possibility that the access value designates some other existing object.

13.11.3 Default Storage Pools

Replace paragraph 1: [8652/0164]

by:

Pragma and aspect Default_Storage_Pool specify the storage pool that will be used in the absence of an explicit specification of a storage pool or storage size for an access type.

Replace paragraph 3.1: [8652/0164]

storage_pool_indicator ::= *storage_pool*_name | **null**

by:

storage_pool_indicator ::= *storage_pool*_name | **null** | Standard

Insert after paragraph 4: [8652/0164]

The *storage_pool*_name shall denote a variable.

the new paragraph:

The Standard storage_pool_indicator is an identifier specific to a pragma (see 2.8) and does not denote any declaration. If the storage_pool_indicator is Standard, then there shall not be a declaration with defining_identifier Standard that is immediately visible at the point of the pragma, other than package Standard itself.

Replace paragraph 4.1: [8652/0164]

If the pragma is used as a configuration pragma, the storage_pool_indicator shall be **null**, and it defines the *default pool* to be **null** within all applicable compilation units (see 10.1.5), except within the immediate scope of another pragma Default_Storage_Pool. Otherwise, the pragma occurs immediately within a sequence of declarations, and it defines the default pool within the immediate scope of the pragma to be either **null** or the pool denoted by the *storage_pool*_name, except within the immediate scope of a later pragma Default_Storage_Pool. Thus, an inner pragma overrides an outer one.

by:

If the pragma is used as a configuration pragma, the storage_pool_indicator shall be either **null** or Standard, and it defines the *default pool* to be the given storage_pool_indicator within all applicable compilation units (see 10.1.5), except within the immediate scope of another pragma Default_Storage_Pool. Otherwise, the pragma occurs immediately within a sequence of declarations, and it defines the default pool within the immediate scope of the pragma to be the given storage_pool_indicator, except within the immediate scope of a later pragma Default_Storage_Pool. Thus, an inner pragma overrides an outer one.

Replace paragraph 5: [8652/0164; 8652/0183]

The language-defined aspect Default_Storage_Pool may be specified for a generic instance; it defines the default pool for access types within an instance. The expected type for the Default_Storage_Pool aspect is Root_Storage_Pool'Class. The aspect_definition must be a name that denotes a variable. This aspect overrides any Default_Storage_Pool pragma that might apply to the generic unit; if the aspect is not specified, the default pool of the instance is that defined for the generic unit.

by:

The language-defined aspect Default_Storage_Pool may be specified for a generic instance; it defines the default pool for access types within an instance.

The Default_Storage_Pool aspect may be specified as Standard, which is an identifier specific to an aspect (see 13.1.1) and defines the default pool to be Standard. In this case, there shall not be a declaration with defining_identifier Standard that is immediately visible at the point of the aspect specification, other than package Standard itself.

Otherwise, the expected type for the Default_Storage_Pool aspect is Root_Storage_Pool'Class and the aspect_definition shall be a name that denotes a variable. This aspect overrides any Default_Storage_Pool pragma that might apply to the generic unit; if the aspect is not specified, the default pool of the instance is that defined for the generic unit.

The effect of specifying the aspect Default_Storage_Pool on an instance of a language-defined generic unit is implementation-defined.

Replace paragraph 6.2: [8652/0164]

- If the default pool is nonnull, the Storage_Pool attribute is that pool.

by:

- If the default pool is neither **null** nor Standard, the Storage_Pool attribute is that pool.

Replace paragraph 6.3: [8652/0164]

Otherwise, there is no default pool; the standard storage pool is used for the type as described in 13.11.

by:

Otherwise (including when the default pool is specified as Standard), the standard storage pool is used for the type as described in 13.11.

13.11.4 Storage Subpools

Replace paragraph 20: [8652/0184]

Each subpool *belongs* to a single storage pool (which will always be a pool that supports subpools). An access to the pool that a subpool belongs to can be obtained by calling Pool_of_Subpool with the subpool handle. Set_Pool_of_Subpool causes the subpool of the subpool handle to belong to the given pool; this is intended to be called from subpool constructors like Create_Subpool. Set_Pool_of_Subpool propagates Program_Error if the subpool already belongs to a pool.

by:

Each subpool *belongs* to a single storage pool (which will always be a pool that supports subpools). An access to the pool that a subpool belongs to can be obtained by calling Pool_of_Subpool with the subpool handle. Set_Pool_of_Subpool causes the subpool of the subpool handle to belong to the given pool; this is intended to be called from subpool constructors like Create_Subpool. Set_Pool_of_Subpool propagates Program_Error if the subpool already belongs to a pool. If Set_Pool_of_Subpool has not yet been called for a subpool, Pool_of_Subpool returns **null**.

Insert after paragraph 31: [8652/0185]

Unless overridden, Default_Subpool_for_Pool propagates Program_Error.

the new paragraph:

Erroneous Execution

If Allocate_From_Subpool does not meet one or more of the requirements on the Allocate procedure as given in the Erroneous Execution rules of 13.11, then the program execution is erroneous.

13.11.5 Subpool Reclamation

Insert after paragraph 7: [8652/0182]

- Any of the objects allocated from the subpool that still exist are finalized in an arbitrary order;

the new paragraph:

- All of the objects allocated from the subpool cease to exist;

13.11.6 Storage Subpool Example

Replace paragraph 11: [8652/0186]

```
type Mark_Release_Pool_Type (Pool_Size : Storage_Count) is new
   Subpools.Root_Storage_Pool_With_Subpools with record
   Storage         : Storage_Array (0 .. Pool_Size-1);
   Next_Allocation : Storage_Count := 0;
   Markers         : Subpool_Array;
   Current_Pool    : Subpool_Indexes := 1;
end record;
```

by:

```
type Mark_Release_Pool_Type (Pool_Size : Storage_Count) is new
   Subpools.Root_Storage_Pool_With_Subpools with record
   Storage         : Storage_Array (0 .. Pool_Size);
   Next_Allocation : Storage_Count := 0;
   Markers         : Subpool_Array;
   Current_Pool    : Subpool_Indexes := 1;
end record;
```

Replace paragraph 28: [8652/0126]

```
   -- Correct the alignment if necessary:
   Pool.Next_Allocation := Pool.Next_Allocation +
      ((-Pool.Next_Allocation) mod Alignment);
   if Pool.Next_Allocation + Size_In_Storage_Elements >
      Pool.Pool_Size then
      raise Storage_Error; -- Out of space.
   end if;
   Storage_Address := Pool.Storage (Pool.Next_Allocation)'Address;
   Pool.Next_Allocation :=
      Pool.Next_Allocation + Size_In_Storage_Elements;
end Allocate_From_Subpool;
```

by:

```
   -- Check for the maximum supported alignment, which is the alignment of the storage area:
   if Alignment > Pool.Storage'Alignment then
      raise Program_Error;
   end if;
   -- Correct the alignment if necessary:
   Pool.Next_Allocation := Pool.Next_Allocation +
      ((-Pool.Next_Allocation) mod Alignment);
   if Pool.Next_Allocation + Size_In_Storage_Elements >
      Pool.Pool_Size then
      raise Storage_Error; -- Out of space.
   end if;
   Storage_Address := Pool.Storage (Pool.Next_Allocation)'Address;
   Pool.Next_Allocation :=
      Pool.Next_Allocation + Size_In_Storage_Elements;
end Allocate_From_Subpool;
```

13.13.2 Stream-Oriented Attributes

Replace paragraph 38: [8652/0177; 8652/0187]

The stream-oriented attributes may be specified for any type via an attribute_definition_clause. The subprogram name given in such a clause shall statically denote a subprogram that is not an abstract subprogram. Furthermore, if a stream-oriented attribute is specified for an interface type by an attribute_definition_clause, the subprogram name given in the clause shall statically denote a null procedure.

by:

The stream-oriented attributes may be specified for any type via an attribute_definition_clause. Alternatively, each of the specific stream-oriented attributes may be specified using an aspect_specification on any type_declaration, with the aspect name being the corresponding attribute name. Each of the class-wide stream-oriented attributes may be specified using an aspect_specification for a tagged type *T* using the name of the stream-oriented attribute followed by 'Class; such class-wide aspects do not apply to other descendants of *T*.

The subprogram name given in such an attribute_definition_clause or aspect_specification shall statically denote a subprogram that is not an abstract subprogram. Furthermore, if a specific stream-oriented attribute is specified for an interface type, the subprogram name given in the attribute_definition_clause or aspect_specification shall statically denote a null procedure.

Replace paragraph 49: [8652/0188]

An attribute_reference for one of the stream-oriented attributes is illegal unless the attribute is available at the place of the attribute_reference. Furthermore, an attribute_reference for *T*'Input is illegal if *T* is an abstract type.

by:

An attribute_reference for one of the stream-oriented attributes is illegal unless the attribute is available at the place of the attribute_reference. Furthermore, an attribute_reference for *T*'Input is illegal if *T* is an abstract type. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

Unless inherited from a parent type, if any, for an untagged type having a task, protected, or explicitly limited record part, the default implementation of each of the Read, Write, Input, and Output attributes raises Program_Error and performs no other action.

13.14 Freezing Rules

Replace paragraph 3: [8652/0189]

The end of a declarative_part, protected_body, or a declaration of a library package or generic library package, causes *freezing* of each entity and profile declared within it, except for incomplete types. A noninstance body other than a renames-as-body causes freezing of each entity and profile declared before it within the same declarative_part that is not an incomplete type; it only causes freezing of an incomplete type if the body is within the immediate scope of the incomplete type.

by:

The end of a declarative_part, protected_body, or a declaration of a library package or generic library package, causes *freezing* of each entity and profile declared within it, except for incomplete types. A proper_body, body_stub, or entry_body causes freezing of each entity and profile declared before it within the same declarative_part that is not an incomplete type; it only causes freezing of an incomplete type if the body is within the immediate scope of the incomplete type.

Insert after paragraph 5: [8652/0189; 8652/0190]

- The occurrence of a generic_instantiation causes freezing, except that a name which is a generic actual parameter whose corresponding generic formal parameter is a formal incomplete type (see 12.5.1) does not cause freezing. In addition, if a parameter of the instantiation is defaulted, the default_expression or default_name for that parameter causes freezing.

the new paragraphs:

- At the occurrence of an expression_function_declaration that is a completion, the return expression of the expression function causes freezing.

- At the occurrence of a renames-as-body whose *callable_entity*_name denotes an expression function, the return expression of the expression function causes freezing.

Replace paragraph 5.1: [8652/0132]

- At the occurrence of an expression_function_declaration that is a completion, the expression of the expression function causes freezing.

by:

- At the occurrence of an expression_function_declaration that is a completion, the return expression of the expression function causes freezing.

Replace paragraph 5.2: [8652/0132]

- At the occurrence of a renames-as-body whose *callable_entity*_name denotes an expression function, the expression of the expression function causes freezing.

by:

- At the occurrence of a renames-as-body whose *callable_entity*_name denotes an expression function, the return expression of the expression function causes freezing.

Replace paragraph 8: [8652/0132]

A static expression (other than within an aspect_specification) causes freezing where it occurs. An object name or nonstatic expression causes freezing where it occurs, unless the name or expression is part of a default_expression, a default_name, the expression of an expression function, an aspect_specification, or a per-object expression of a component's constraint, in which case, the freezing occurs later as part of another construct or at the freezing point of an associated entity.

by:

A static expression (other than within an aspect_specification) causes freezing where it occurs. An object name or nonstatic expression causes freezing where it occurs, unless the name or expression is part of a default_expression, a default_name, the return expression of an expression function, an aspect_specification, or a per-object expression of a component's constraint, in which case, the freezing occurs later as part of another construct or at the freezing point of an associated entity.

Replace paragraph 10.1: [8652/0132]

- At the place where a function call causes freezing, the profile of the function is frozen. Furthermore, if a parameter of the call is defaulted, the default_expression for that parameter causes freezing. If the function call is to an expression function, the expression of the expression function causes freezing.

by:

- At the place where a function call causes freezing, the profile of the function is frozen. Furthermore, if a parameter of the call is defaulted, the default_expression for that parameter causes freezing. If the function call is to an expression function, the return expression of the expression function causes freezing.

Replace paragraph 10.2: [8652/0132]

- At the place where a generic_instantiation causes freezing of a callable entity, the profile of that entity is frozen unless the formal subprogram corresponding to the callable entity has a parameter or result of a formal untagged incomplete type; if the callable entity is an expression function, the expression of the expression function causes freezing.

by:

- At the place where a generic_instantiation causes freezing of a callable entity, the profile of that entity is frozen unless the formal subprogram corresponding to the callable entity has a parameter or result of a formal untagged incomplete type; if the callable entity is an expression function, the return expression of the expression function causes freezing.

Replace paragraph 10.3: [8652/0132]

- At the place where a use of the Access or Unchecked_Access attribute whose prefix denotes an expression function causes freezing, the expression of the expression function causes freezing.

by:

- At the place where a use of the Access or Unchecked_Access attribute whose prefix denotes an expression function causes freezing, the return expression of the expression function causes freezing.

Annex A: Predefined Language Environment

Replace paragraph 3: [8652/0191; 8652/0192; 8652/0125]

The implementation shall ensure that each language-defined subprogram is reentrant in the sense that concurrent calls on the same subprogram perform as specified, so long as all parameters that could be passed by reference denote nonoverlapping objects.

by:

The implementation shall ensure that each language-defined subprogram is reentrant in the sense that concurrent calls on any language-defined subprogram perform as specified, so long as all objects that are denoted by parameters that could be passed by reference or designated by parameters of an access type are nonoverlapping.

For the purpose of determining whether concurrent calls on text input-output subprograms are required to perform as specified above, when calling a subprogram within Text_IO or its children that implicitly operates on one of the default input-output files, the subprogram is considered to have a parameter of Current_Input or Current_Output (as appropriate).

A.4.11 String Encoding

Replace paragraph 54: [8652/0193]

- By a Decode function when a UTF encoded string contains an invalid encoding sequence.

by:

- By a Convert or Decode function when a UTF encoded string contains an invalid encoding sequence.

Replace paragraph 55: [8652/0193]

- By a Decode function when the expected encoding is UTF-16BE or UTF-16LE and the input string has an odd length.

by:

- By a Convert or Decode function when the expected encoding is UTF-16BE or UTF-16LE and the input string has an odd length.

A.8.1 The Generic Package Sequential_IO

Insert after paragraph 10: [8652/0194]

```
function Is_Open(File : in File_Type) return Boolean;
```

the new paragraph:

```
procedure Flush (File : in File_Type);
```

A.8.2 File Management

Insert after paragraph 28: [8652/0194]

Returns True if the file is open (that is, if it is associated with an external file); otherwise, returns False.

the new paragraphs:

```
procedure Flush(File : in File_Type);
```

The Flush procedure synchronizes the external file with the internal file (by flushing any internal buffers) without closing the file. For a direct file, the current index is unchanged; for a stream file (see A.12.1), the current position is unchanged.

The exception Status_Error is propagated if the file is not open. The exception Mode_Error is propagated if the mode of the file is In_File.

A.8.4 The Generic Package Direct_IO

Insert after paragraph 10: [8652/0194]

```
function Is_Open(File : in File_Type) return Boolean;
```

the new paragraph:

```
procedure Flush (File : in File_Type);
```

A.10.3 Default Input, Output, and Error Files

Replace paragraph 21: [8652/0194]

The effect of Flush is the same as the corresponding subprogram in {Sequential_IO (see A.8.2)}[Streams.Stream_IO (see A.12.1)]. If File is not explicitly specified, Current_Output is used.

by:

The effect of Flush is the same as the corresponding subprogram in Sequential_IO (see A.8.2). If File is not explicitly specified, Current_Output is used.

A.12.1 The Package Streams.Stream_IO

Replace paragraph 5: [8652/0195]

```
type File_Type is limited private;
```

by:

```
type File_Type is limited private;
pragma Preelaborable_Initialization(File_Type);
```

Replace paragraph 28: [8652/0194]

The subprograms given in subclause A.8.2 for the control of external files (Create, Open, Close, Delete, Reset, Mode, Name, Form, and Is_Open) are available for stream files.

by:

The subprograms given in subclause A.8.2 for the control of external files (Create, Open, Close, Delete, Reset, Mode, Name, Form, Is_Open, and Flush) are available for stream files.

Delete paragraph 28.6: [8652/0194]

The Flush procedure synchronizes the external file with the internal file (by flushing any internal buffers) without closing the file or changing the position. Mode_Error is propagated if the mode of the file is In_File.

A.18 Containers

Insert after paragraph 5: [8652/0196]

When a formal function is used to provide an ordering for a container, it is generally required to define a strict weak ordering. A function "<" defines a *strict weak ordering* if it is irreflexive, asymmetric, transitive, and in addition, if $x < y$ for any values x and y, then for all other values z, $(x < z)$ or $(z < y)$.

the new paragraphs:

Static Semantics

Certain subprograms declared within instances of some of the generic packages presented in this clause are said to *perform indefinite insertion*. These subprograms are those corresponding (in the sense of the copying described in subclause 12.3) to subprograms that have formal parameters of a generic formal indefinite type and that are identified as performing indefinite insertion in the subclause defining the generic package.

If a subprogram performs indefinite insertion, then certain run-time checks are performed as part of a call to the subprogram; if any of these checks fail, then the resulting exception is propagated to the caller and the container is not modified by the call. These checks are performed for each parameter corresponding (in the sense of the copying described in 12.3) to a parameter in the corresponding generic whose type is a generic formal indefinite type. The checks performed for a given parameter are those checks explicitly specified in subclause 4.8 that would be performed as part of the evaluation of an initialized allocator whose access type is declared immediately within the instance, where:

- the value of the qualified_expression is that of the parameter; and
- the designated subtype of the access type is the subtype of the parameter; and

- finalization of the collection of the access type has started if and only if the finalization of the instance has started.

A.18.2 The Generic Package Containers.Vectors

Replace paragraph 97.1: [8652/0197]

When tampering with cursors is *prohibited* for a particular vector object *V*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *V*, leaving *V* unmodified. Similarly, when tampering with elements is *prohibited* for a particular vector object *V*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *V* (or tamper with the cursors of *V*), leaving *V* unmodified.

by:

When tampering with cursors is *prohibited* for a particular vector object *V*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *V*, leaving *V* unmodified. Similarly, when tampering with elements is *prohibited* for a particular vector object *V*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *V* (or tamper with the cursors of *V*), leaving *V* unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

Replace paragraph 168: [8652/0126]

```
procedure Prepend (Container : in out Vector;
                   New_Item  : in     Vector;
                   Count     : in     Count_Type := 1);
```

by:

```
procedure Prepend (Container : in out Vector;
                   New_Item  : in     Vector);
```

A.18.3 The Generic Package Containers.Doubly_Linked_Lists

Replace paragraph 69.1: [8652/0197]

When tampering with cursors is *prohibited* for a particular list object *L*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *L*, leaving *L* unmodified. Similarly, when tampering with elements is *prohibited* for a particular list object *L*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *L* (or tamper with the cursors of *L*), leaving *L* unmodified.

by:

When tampering with cursors is *prohibited* for a particular list object *L*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *L*, leaving *L* unmodified. Similarly, when tampering with elements is *prohibited* for a particular list object *L*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *L* (or tamper with the cursors of *L*), leaving *L* unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

A.18.4 Maps

Replace paragraph 15.1: [8652/0197]

When tampering with cursors is *prohibited* for a particular map object *M*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *M*, leaving *M* unmodified. Similarly, when tampering with elements is *prohibited* for a particular map object *M*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *M* (or tamper with the cursors of *M*), leaving *M* unmodified.

by:

When tampering with cursors is *prohibited* for a particular map object *M*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *M*, leaving *M* unmodified. Similarly, when tampering with elements is *prohibited* for a particular map object *M*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *M* (or tamper with the cursors of *M*), leaving *M* unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

A.18.7 Sets

Replace paragraph 14.1: [8652/0197]

When tampering with cursors is *prohibited* for a particular set object *S*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *S*, leaving *S* unmodified. Similarly, when tampering with elements is *prohibited* for a particular set object *S*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *S* (or tamper with the cursors of *S*), leaving *S* unmodified.

by:

When tampering with cursors is *prohibited* for a particular set object *S*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *S*, leaving *S* unmodified. Similarly, when tampering with elements is *prohibited* for a particular set object *S*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *S* (or tamper with the cursors of *S*), leaving *S* unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

A.18.10 The Generic Package Containers.Multiway_Trees

Replace paragraph 2: [8652/0198; 8652/0125]

A multiway tree container object manages a tree of internal *nodes*, each of which contains an element and pointers to the parent, first child, last child, next (successor) sibling, and previous (predecessor) sibling internal nodes. A cursor designates a particular node within a tree (and by extension the element contained in that node, if any). A cursor keeps designating the same node (and element) as long as the node is part of the container, even if the node is moved within the container.

by:

A multiway tree container object manages a tree of *nodes*, consisting of a *root node* and a set of *internal nodes*; each internal node contains an element and pointers to the parent, first child, last child, next (successor) sibling, and previous (predecessor) sibling internal nodes. A cursor designates a particular node within a tree (and by extension the element contained in that node, if any). A cursor keeps designating the same node (and element) as long as the node is part of the container, even if the node is moved within the container.

Replace paragraph 3: [8652/0198]

A *subtree* is a particular node (which *roots the subtree*) and all of its child nodes (including all of the children of the child nodes, recursively). There is a special node, the *root*, which is always present and has neither an associated element value nor any parent node. The root node provides a place to add nodes to an otherwise empty tree and represents the base of the tree.

by:

A *subtree* is a particular node (which *roots the subtree*) and all of its child nodes (including all of the children of the child nodes, recursively). The root node is always present and has neither an associated element value nor any parent node; it has pointers to its first child and its last child, if any. The root node provides a place to add nodes to an otherwise empty tree and represents the base of the tree.

Replace paragraph 90: [8652/0197]

When tampering with cursors is *prohibited* for a particular tree object *T*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *T*, leaving *T* unmodified. Similarly, when tampering with elements is *prohibited* for a particular tree object *T*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *T* (or tamper with the cursors of *T*), leaving *T* unmodified.

by:

When tampering with cursors is *prohibited* for a particular tree object *T*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of *T*, leaving *T* unmodified. Similarly, when tampering with elements is *prohibited* for a particular tree object *T*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of *T* (or tamper with the cursors of *T*), leaving *T* unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

Replace paragraph 153: [8652/0199]

Iterate calls Process.**all** with a cursor that designates each element in Container, starting with the root node and proceeding in a depth-first order. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

by:

Iterate calls Process.**all** with a cursor that designates each element in Container, starting from the root node and proceeding in a depth-first order. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

Replace paragraph 155: [8652/0199]

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Iterate_Subtree calls Process.**all** with a cursor that designates each element in the subtree rooted by the node designated by Position, starting with the node designated by Position and proceeding in a depth-first order. Tampering with the cursors of the tree that contains the element designated by Position is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

by:

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Iterate_Subtree calls Process.**all** with a cursor that designates each element in the subtree rooted by the node designated by Position, starting from the node designated by Position and proceeding in a depth-first order. Tampering with the cursors of the tree that contains the element designated by Position is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

Replace paragraph 157: [8652/0199]

Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the root node and proceeding in a depth-first order. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

by:

Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in Container, starting from the root node and proceeding in a depth-first order. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

Replace paragraph 159: [8652/0199]

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Iterate_Subtree returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in the subtree rooted by the node designated by Position, starting with the node designated by Position and proceeding in a depth-first order. If Position equals No_Element, then Constraint_Error is propagated. Tampering with the cursors of the container that contains the node designated by Position is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

by:

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Iterate_Subtree returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in the subtree rooted by the node designated by Position, starting from the node designated by Position and proceeding in a depth-first order. If Position equals No_Element, then Constraint_Error is propagated. Tampering with the cursors of the container that contains the node designated by Position is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

A.18.11 The Generic Package Containers.Indefinite_Vectors

Insert after paragraph 8: [8652/0196]

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

the new paragraph:

- The operations "&", Append, Insert, Prepend, Replace_Element, and To_Vector that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

A.18.12 The Generic Package Containers.Indefinite_Doubly_Linked_Lists

Insert after paragraph 7: [8652/0196]

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

the new paragraph:

- The operations Append, Insert, Prepend, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

A.18.13 The Generic Package Containers.Indefinite_Hashed_Maps

Insert after paragraph 8: [8652/0196]

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

the new paragraph:

- The operations Include, Insert, Replace, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

A.18.14 The Generic Package Containers.Indefinite_Ordered_Maps

Insert after paragraph 8: [8652/0196]

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

the new paragraph:

- The operations Include, Insert, Replace, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

A.18.15 The Generic Package Containers.Indefinite_Hashed_Sets

Insert after paragraph 4: [8652/0196]

- The actual Element parameter of access subprogram Process of Update_Element_Preserving_Key may be constrained even if Element_Type is unconstrained.

the new paragraph:

- The operations Include, Insert, Replace, Replace_Element, and To_Set that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

A.18.16 The Generic Package Containers.Indefinite_Ordered_Sets

Insert after paragraph 4: [8652/0196]

- The actual Element parameter of access subprogram Process of Update_Element_Preserving_Key may be constrained even if Element_Type is unconstrained.

the new paragraph:

- The operations Include, Insert, Replace, Replace_Element, and To_Set that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

A.18.17 The Generic Package Containers.Indefinite_Multiway_Trees

Insert after paragraph 7: [8652/0196]

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

the new paragraph:

- The operations Append_Child, Insert_Child, Prepend_Child, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

A.18.18 The Generic Package Containers.Indefinite_Holders

Replace paragraph 35: [8652/0197]

When tampering with the element is *prohibited* for a particular holder object *H*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the element of *H*, leaving *H* unmodified.

by:

When tampering with the element is *prohibited* for a particular holder object *H*, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the element of *H*, leaving *H* unmodified. These checks are made before any other defined behavior of the body of the language-defined subprogram.

Replace paragraph 39: [8652/0196]

Returns a nonempty holder containing an element initialized to New_Item.

by:

Returns a nonempty holder containing an element initialized to New_Item. To_Holder performs indefinite insertion (see A.18).

Replace paragraph 47: [8652/0196]

Replace_Element assigns the value New_Item into Container, replacing any preexisting content of Container. Container is not empty after a successful call to Replace_Element.

by:

Replace_Element assigns the value New_Item into Container, replacing any preexisting content of Container; Replace_Element performs indefinite insertion (see A.18). Container is not empty after a successful call to Replace_Element.

A.18.25 The Generic Package Containers.Bounded_Multiway_Trees

Replace paragraph 10: [8652/0118]

```
function Copy (Source : Tree; Capacity : Count_Type := 0)
   return List;
```

by:

```
function Copy (Source : Tree; Capacity : Count_Type := 0)
   return Tree;
```

A.18.26 Array Sorting

Replace paragraph 9.2: [8652/0118]

```
generic
   type Index_Type is (<>);
   with function Before (Left, Right : Index_Type) return Boolean;
   with procedure Swap (Left, Right : Index_Type);
procedure Ada.Containers.Generic_Sort
```

```
      (First, Last : Index_Type'Base);
pragma Pure(Ada.Containers.Generic_Sort);
```

by:

```
generic
   type Index_Type is (<>);
   with function Before (Left, Right : Index_Type) return Boolean;
   with procedure Swap (Left, Right : in Index_Type);
procedure Ada.Containers.Generic_Sort
      (First, Last : Index_Type'Base);
pragma Pure(Ada.Containers.Generic_Sort);
```

A.18.32 Example of Container Use

Replace paragraph 29: [8652/0126]

```
for C in G (Next).Iterate loop
   declare
      E : Edge renames G (Next)(C).all;
   begin
      if not Reached(E.To) then
         ...
      end if;
   end;
end loop;
```

by:

```
for C in G (Next).Iterate loop
   declare
      E : Edge renames G (Next)(C);
   begin
      if not Reached(E.To) then
         ...
      end if;
   end;
end loop;
```

Replace paragraph 31: [8652/0126]

```
declare
   L : Adjacency_Lists.List renames G (Next);
   C : Adjacency_Lists.Cursor := L.First;
begin
   while Has_Element (C) loop
      declare
         E : Edge renames L(C).all;
      begin
         if not Reached(E.To) then
            ...
         end if;
      end;
      C := L.Next (C);
   end loop;
end;
```

by:

```
declare
   L : Adjacency_Lists.List renames G (Next);
   C : Adjacency_Lists.Cursor := L.First;
begin
   while Has_Element (C) loop
      declare
         E : Edge renames L(C);
      begin
         if not Reached(E.To) then
            ...
```

```
            end if;
         end;
         C := L.Next (C);
      end loop;
   end;
```

A.19 The Package Locales

Replace paragraph 4: [8652/0200]

```
type Language_Code is array (1 .. 3) of Character range 'a' .. 'z';
type Country_Code is array (1 .. 2) of Character range 'A' .. 'Z';
```

by:

```
type Language_Code is new String (1 .. 3)
   with Dynamic_Predicate =>
     (for all E of Language_Code => E in 'a' .. 'z');
type Country_Code is new String (1 .. 2)
   with Dynamic_Predicate =>
     (for all E of Country_Code  => E in 'A' .. 'Z');
```

Annex B: Interface to Other Languages

B.1 Interfacing Aspects

Insert after paragraph 14: [8652/0201]

- Convention *L* has been specified for T, and T is *eligible for convention L*; that is:

the new paragraph:

- T is an enumeration type such that all internal codes (whether assigned by default or explicitly) are within an implementation-defined range that includes at least the range of values 0 .. 2**15–1;

Replace paragraph 41: [8652/0201]

For each supported convention *L* other than Intrinsic, an implementation should support specifying the Import and Export aspects for objects of *L*-compatible types and for subprograms, and the Convention aspect for *L*-eligible types and for subprograms, presuming the other language has corresponding features. Specifying the Convention aspect need not be supported for scalar types.

by:

For each supported convention *L* other than Intrinsic, an implementation should support specifying the Import and Export aspects for objects of *L*-compatible types and for subprograms, and the Convention aspect for *L*-eligible types and for subprograms, presuming the other language has corresponding features. Specifying the Convention aspect need not be supported for scalar types, other than enumeration types whose internal codes fall within the range 0 .. 2**15–1.

Replace paragraph 50: [8652/0126]

Example of interfacing pragmas:

by:

Example of interfacing aspects:

B.3 Interfacing with C and C++

Replace paragraph 1: [8652/0202]

The facilities relevant to interfacing with the C language and the corresponding subset of the C++ language are the package Interfaces.C and its children, and support for specifying the Convention aspect with *convention*_identifiers C and C_Pass_By_Copy.

by:

The facilities relevant to interfacing with the C language and the corresponding subset of the C++ language are the package Interfaces.C and its children, and support for specifying the Convention aspect with *convention*_identifiers C, C_Pass_By_Copy, and any of the C_Variadic_*n* conventions described below.

Insert after paragraph 60.15: [8652/0202]

If a type is C_Pass_By_Copy-compatible, then it is also C-compatible.

the new paragraph:

The identifiers C_Variadic_0, C_Variadic_1, C_Variadic_2, and so on are *convention*_identifiers. These conventions are said to be *C_Variadic*. The convention C_Variadic_*n* is the calling convention for a variadic C function taking *n* fixed parameters and then a variable number of additional parameters. The C_Variadic_*n* convention shall only be specified as the convention aspect for a subprogram, or for an access-to-subprogram type, having at least *n* parameters. A type is compatible with a C_Variadic convention if and only if the type is C-compatible.

Insert after paragraph 65: [8652/0201]

- An Ada function corresponds to a non-void C function.

the new paragraph:

- An Ada enumeration type corresponds to a C enumeration type with corresponding enumeration literals having the same internal codes, provided the internal codes fall within the range of the C int type.

Replace paragraph 75: [8652/0202]

A C function that takes a variable number of arguments can correspond to several Ada subprograms, taking various specific numbers and types of parameters.

by:

A variadic C function can correspond to several Ada subprograms, taking various specific numbers and types of parameters.

Annex C: Systems Programming

C.5 Aspect Discard_Names

Replace the title: [8652/0203]

Pragma Discard_Names

by:

Aspect Discard_Names

Replace paragraph 1: [8652/0203]

A pragma Discard_Names may be used to request a reduction in storage used for the names of certain entities.

by:

Specifying the aspect Discard_Names can be used to request a reduction in storage used for the names of entities with runtime name text.

Static Semantics

An entity with *runtime name text* is a nonderived enumeration first subtype, a tagged first subtype, or an exception.

For an entity with runtime name text, the following language-defined representation aspect may be specified:

Discard_Names

The type of aspect Discard_Names is Boolean. If directly specified, the aspect_definition shall be a static expression. If not specified (including by inheritance), the aspect is False.

Replace paragraph 5: [8652/0203]

The local_name (if present) shall denote a nonderived enumeration [first] subtype, a tagged [first] subtype, or an exception. The pragma applies to the type or exception. Without a local_name, the pragma applies to all such entities declared after the pragma, within the same declarative region. Alternatively, the pragma can be used as a configuration pragma. If the pragma applies to a type, then it applies also to all descendants of the type.

by:

The local_name (if present) shall denote an entity with runtime name text. The pragma specifies that the aspect Discard_Names for the type or exception has the value True. Without a local_name, the pragma specifies that all entities with runtime name text declared after the pragma, within the same declarative region have the value True for aspect Discard_Names. Alternatively, the pragma can be used as a configuration pragma. If the configuration pragma Discard_Names applies to a compilation unit, all entities with runtime name text declared in the compilation unit have the value True for the aspect Discard_Names.

Replace paragraph 7: [8652/0203]

If the pragma applies to an enumeration type, then the semantics of the Wide_Wide_Image and Wide_Wide_Value attributes are implementation defined for that type; the semantics of Image, Wide_Image, Value, and Wide_Value are still defined in terms of Wide_Wide_Image and Wide_Wide_Value. In addition, the semantics of Text_IO.Enumeration_IO are implementation defined. If the pragma applies to a tagged type, then the semantics of the Tags.Wide_Wide_Expanded_Name function are implementation defined for that type; the semantics of Tags.Expanded_Name and Tags.Wide_Expanded_Name are still defined in terms of Tags.Wide_Wide_Expanded_Name. If the pragma applies to an exception, then the semantics of the Exceptions.Wide_Wide_Exception_Name function are implementation defined for that exception; the semantics of Exceptions.Exception_Name and Exceptions.Wide_Exception_Name are still defined in terms of Exceptions.Wide_Wide_Exception_Name.

by:

If the aspect Discard_Names is True for an enumeration type, then the semantics of the Wide_Wide_Image and Wide_Wide_Value attributes are implementation defined for that type; the semantics of Image, Wide_Image, Value, and Wide_Value are still defined in terms of Wide_Wide_Image and Wide_Wide_Value. In addition, the semantics of Text_IO.Enumeration_IO are implementation defined. If the aspect Discard_Names is True for a tagged type, then the semantics of the Tags.Wide_Wide_Expanded_Name function are implementation defined for that type; the semantics of Tags.Expanded_Name and Tags.Wide_Expanded_Name are still defined in terms of Tags.Wide_Wide_Expanded_Name. If the aspect Discard_Names is True for an exception, then the semantics of the Exceptions.Wide_Wide_Exception_Name