Information technology —Programming languages — Ada

AMENDMENT 2

Technologies de l'information —Langages de programmation — Ada

AMENDEMENT 2

Amendment 2 to International Standard ISO/IEC 8652:1995 was prepared by AXE Consultants.

© 2008, AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Compiled copies of standard library units and examples need not contain this copyright notice so long as the notice is included in all copies of the source code and documentation. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

Disclaimer:

This document is a draft of a possible second amendment to Ada 95 (International Standard ISO/IEC 8652:1995). This draft contains only proposals substantially approved by the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group (ARG). Many other important proposals are under consideration by the ARG. Neither the ARG nor any other group has determined which, if any, of these proposals will be included in the amendment. Any proposal may be substantially changed or withdrawn before this document begins standardization, and other proposals may be added. This document is not an official publication or work product of the ARG.

Introduction

International Standard ISO/IEC 8652:1995 defines the Ada programming language.

This amendment modifies Ada by making changes and additions that improve:

- The safety of applications written in Ada;
- The portability of applications written in Ada;
- Interoperability with other languages and systems; and
- Accessibility and ease of transition from idioms in other programming and modeling languages.

This amendment incorporates the following major additions to the International Standard:

- Generalizing extended return statements (see 6.5);
- Wording and terminology improvements for limited types (see 7.5 and 7.6); and
- Additional kinds of and additional operations for containers (see A.18).

This Amendment is organized by sections corresponding to those in the International Standard. These sections include wording changes and additions to the International Standard. Clause and subclause headings are given for each clause that contains a wording change. Clauses and subclauses that do not contain any change or addition are omitted.

For each change, an *anchor* paragraph from the International Standard (as corrected by Technical Corrigendum 1) 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 International Standard - this font for text, this font for syntax, `and this font for Ada source code.`

Section 1: General

1.1.2 Structure

Replace paragraph 12: [AI05-0004-1]

Annex M, "Implementation-Defined Characteristics"

by:

Annex M, "Summary of Documentation Requirements"

1.1.4 Method of Description and Syntax Notation

Replace paragraph 14.1: [AI05-0004-1]

The delimiters, compound delimiters, reserved words, and numeric_literals are exclusively made of the characters whose code position is between 16#20# and 16#7E#, inclusively. The special characters for which names are defined in this International Standard (see 2.1) belong to the same range. For example, the character E in the definition of exponent is the character whose name is "LATIN CAPITAL LETTER E", not "GREEK CAPITAL LETTER EPSILON".

by:

The delimiters, compound delimiters, reserved words, and numeric_literals are exclusively made of the characters whose code position is between 16#20# and 16#7E#, inclusively. The special characters for which names are defined in this International Standard (see 2.1) belong to the same range. For example, the character E in the definition of exponent is the character whose name is "LATIN CAPITAL LETTER E", not "GREEK CAPITAL LETTER EPSILON".

Section 2: Lexical Elements

2.1 Character Set

Replace paragraph 4: [AI05-0079-1]

The coded representation for characters is implementation defined (it need not be a representation defined within ISO/IEC 10646:2003). A character whose relative code position in its plane is 16#FFFE# or 16#FFFF# is not allowed anywhere in the text of a program.

by:

The coded representation for characters is implementation defined (it need not be a representation defined within ISO/IEC 10646:2003). A character whose relative code position in its plane is 16#FFFE# or 16#FFFF# is not allowed anywhere in the text of a program. The only characters allowed outside of comments are those in categories other_format, format_effector, and graphic_character.

2.2 Lexical Elements, Separators, and Delimiters

Insert after paragraph 7: [AI05-0079-1]

One or more separators are allowed between any two adjacent lexical elements, before the first of each compilation, or after the last. At least one separator is required between an identifier, a reserved word, or a numeric_literal and an adjacent identifier, reserved word, or numeric_literal.

the new paragraph:

One of more other_format characters are allowed anywhere that a separator is; any such characters have no effect on the meaning of an Ada program.

2.3 Identifiers

Replace paragraph 3.1: [AI05-0091-1]

```
identifier_extend ::=
    mark_non_spacing
  | mark_spacing_combining
  | number_decimal
  | punctuation_connector
  | other_format
```

by:

```
identifier_extend ::=
    mark_non_spacing
  | mark_spacing_combining
  | number_decimal
  | punctuation_connector
```

Replace paragraph 4: [AI05-0091-1]

After eliminating the characters in category other_format, an identifier shall not contain two consecutive characters in category punctuation_connector, or end with a character in that category.

by:

An identifier shall not contain two consecutive characters in category punctuation_connector, or end with a character in that category.

Replace paragraph 5: [AI05-0091-1]

Two identifiers are considered the same if they consist of the same sequence of characters after applying the following transformations (in this order):

by:

Two identifiers are considered the same if they consist of the same sequence of characters after converting the characters to upper case.

Delete paragraph 5.2: [AI05-0091-1]

- The remaining sequence of characters is converted to upper case.

Replace paragraph 5.3: [AI05-0091-1]

After applying these transformations, an identifier shall not be identical to a reserved word (in upper case).

by:

After converting to upper case, an identifier shall not be identical to a reserved word (in upper case).

2.8 Pragmas

Replace paragraph 7: [AI05-0100-1]

- At any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", "statement", "clause", or "alternative", or one of the syntactic categories variant or exception_handler; but not in place of such a construct. Also at any place where a compilation_unit would be allowed.

by:

- At any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", "item", "statement", "clause", or "alternative", or one of the syntactic categories variant or exception_handler; but not in place of such a construct if the construct is required, or is part of a list that is required to have at least one such construct, as in a sequence_of_statements.
- At any place where a compilation_unit would be allowed.

2.9 Reserved Words

Replace paragraph 2: [AI05-0091-1]

The following are the *reserved words*. Within a program, some or all of the letters of a reserved word may be in upper case, and one or more characters in category other_format may be inserted within or at the end of the reserved word.

by:

The following are the *reserved words*. Within a program, some or all of the letters of a reserved word may be in upper case.

Section 3: Declarations and Types

3.1 Declarations

Insert after paragraph 7: [AI05-0080-1]

All declarations contain a *definition* for a *view* of an entity. A view consists of an identification of the entity (the entity *of* the view), plus view-specific characteristics that affect the use of the entity through that view (such as mode of access to an object, formal parameter names and defaults for a subprogram, or visibility to components of a type). In most cases, a declaration also contains the definition for the entity itself (a renaming_declaration is an example of a declaration that does not define a new entity, but instead defines a view of an existing entity (see 8.5)).

the new paragraph:

When it is clear from context, the term *object* is used in place of *view of an object*. Similarly, the terms *type* and *subtype* are used in place of *view of a type* and *view of a subtype*, respectively.

3.3 Objects and Named Numbers

Replace paragraph 10: [AI05-0015-1]

- the return object created as the result of evaluating a function_call (or the equivalent operator invocation @mdash see 6.6);

by:

- the return object of a function;

Insert after paragraph 11: [AI05-0003-1]

- the result of evaluating an aggregate;

the new paragraph:

- a qualified_expression whose operand denotes an object;

Replace paragraph 13: [AI05-0054-2]

An object is either a *constant* object or a *variable* object. The value of a constant object cannot be changed between its initialization and its finalization, whereas the value of a variable object can be changed. Similarly, a view of an object is either a *constant* or a *variable*. All views of a constant object are constant. A constant view of a variable object cannot be used to modify the value of the variable. The terms constant and variable by themselves refer to constant and variable views of objects.

by:

An object is either a *constant* object or a *variable* object. Similarly, a view of an object is either a *constant* or a *variable*. All views of a constant elementary object are constant. All views of a constant composite object are constant, except for parts that are of controlled or immutably limited types; variable views of those parts and their subcomponents may exist. In this sense, objects of controlled and immutably limited types are *inherently mutable*. A constant view of an object cannot be used to modify its value. The terms constant and variable by themselves refer to constant and variable views of objects.

Replace paragraph 21: [AI05-0003-1; AI05-0015-1; AI05-0120-1]

- the result of evaluating a function_call or an aggregate;

by:

- the return object declared by an extended_return_statement with the reserved word **constant**;
- the object denoted by a function_call or aggregate;
- the result of evaluating a qualified_expression;
- within the body of a protected function (or a function declared immediately within a protected_body), the current instance of the enclosing protected unit;

Insert after paragraph 23: [AI05-0008-1; AI05-0041-1; AI05-0093-1]

At the place where a view of an object is defined, a *nominal subtype* is associated with the view. The object's *actual subtype* (that is, its subtype) can be more restrictive than the nominal subtype of the view; it always is if the nominal subtype is an *indefinite subtype*. A subtype is an indefinite subtype if it is an unconstrained array subtype, or if it has unknown discriminants or unconstrained discriminants without defaults (see 3.7); otherwise the subtype is a definite subtype (all elementary subtypes are

definite subtypes). A class-wide subtype is defined to have unknown discriminants, and is therefore an indefinite subtype. An indefinite subtype does not by itself provide enough information to create an object; an additional constraint or explicit initialization expression is necessary (see 3.3.1). A component cannot have an indefinite nominal subtype.

the new paragraphs:

A view of a composite object is *known to be constrained* if:

- its nominal subtype is constrained, and is not an untagged partial view; or
- its nominal subtype is indefinite; or
- its type is immutably limited (see 7.5); or
- it is part of a stand-alone constant (including a generic formal object of mode **in**); or
- it is part of a formal parameter of mode **in**; or
- it is part of the object denoted by a function_call or aggregate; or
- it is part of a constant return object of an extended_return_statement; or
- it is a dereference of a pool-specific access type, and there is no ancestor of its type that has a constrained partial view.

For the purposes of determining within a generic body whether an object is known to be constrained:

- if a subtype is a descendant of an untagged generic formal private or derived type, and the subtype is not an unconstrained array subtype, it is not considered indefinite and is considered to have a constrained partial view;
- if a subtype is a descendant of a formal access type, it is not considered pool-specific.

Insert after paragraph 25: [AI05-0054-2]

NOTES

5 A constant cannot be the target of an assignment operation, nor be passed as an **in out** or **out** parameter, between its initialization and finalization, if any.

the new paragraph:

6 The value of a constant object cannot be changed after its initialization, except in some cases where the object has a controlled or immutably limited part (see 7.5, 7.6, and 13.9.1).

3.4 Derived Types and Classes

Replace paragraph 5.1: [AI05-0096-1]

If the reserved word **limited** appears in a derived_type_definition, the parent type shall be a limited type.

by:

If the reserved word **limited** appears in a derived_type_definition, the parent type shall be a limited type. If the parent type is a tagged formal type, then 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.

3.5.1 Enumeration Types

Replace paragraph 6: [AI05-0006-1]

Each enumeration_literal_specification is the explicit declaration of the corresponding *enumeration literal*: it declares a parameterless function, whose defining name is the defining_identifier or defining_character_literal, and whose result type is the enumeration type.

by:

Each enumeration_literal_specification is the explicit declaration of the corresponding *enumeration literal*: it declares a parameterless function, whose defining name is the defining_identifier or defining_character_literal, and whose result subtype is the base subtype of the enumeration type.

3.7 Discriminants

Replace paragraph 10: [AI05-0063-1]

A discriminant_specification for an access discriminant may have a default_expression only in the declaration for a task or protected type, or for a type that is a descendant of an explicitly limited record 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.

by:

A discriminant_specification for an access discriminant may have a default_expression only in the declaration for an immutably limited type (see 7.5). 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.

3.7.1 Discriminant Constraints

Replace paragraph 7: [AI05-0041-1]

A discriminant_constraint is only allowed in a subtype_indication whose subtype_mark denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype. However, in the case of an access subtype, a discriminant_constraint is illegal if the designated type has a partial view that is constrained or, for a general access subtype, has default_expressions for its discriminants. 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. In a generic body, this rule is checked presuming all formal access types of the generic might be general access types, and all untagged discriminated formal types of the generic might have default_expressions for their discriminants.

by:

A discriminant_constraint is only allowed in a subtype_indication whose subtype_mark denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype. However, in the case of an access subtype, a discriminant_constraint is legal only if any dereference of a value of the access type is known to be constrained (see 3.3). 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.

3.8 Record Types

Replace paragraph 11: [AI05-0004-1]

- A name that denotes any component, protected subprogram, or entry is allowed within a representation item that occurs within the declaration of the composite type.

by:

- A name that denotes any component, protected subprogram, or entry is allowed within an operational or representation item that occurs within the declaration of the composite type.

Replace paragraph 13.1: [AI05-0004-1]

If a record_type_declaration includes the reserved word **limited**, the type is called an *explicitly limited record* type.

by:

If a record_type_definition includes the reserved word **limited**, the type is called an *explicitly limited record* type.

3.9.2 Dispatching Operations of Tagged Types

Replace paragraph 2: [AI05-0076-1]

A *call on a dispatching operation* is a call whose name or prefix denotes the declaration of a dispatching operation. A *controlling operand* in a call on a dispatching operation of a tagged type *T* is one whose corresponding formal parameter is of type *T* or is of an anonymous access type with designated type *T*; the corresponding formal parameter is called a *controlling formal parameter*. If the controlling formal parameter is an access parameter, the controlling operand is the object designated by the actual parameter, rather than the actual parameter itself. If the call is to a (primitive) function with result type *T*, then the call has a *controlling result* — the context of the call can control the dispatching. Similarly, if the call is to a function with access result type designating *T*, then the call has a *controlling access result*, and the context can similarly control dispatching.

by:

A *call on a dispatching operation* is a call whose name or prefix denotes the declaration of a dispatching operation. A *controlling operand* in a call on a dispatching operation of a tagged type *T* is one whose corresponding formal parameter is of type *T* or is of an anonymous access type with designated type *T*; the corresponding formal parameter is called a *controlling formal parameter*. If the controlling formal parameter is an access parameter, the controlling operand is the object designated by the actual parameter, rather than the actual parameter itself. If the call is to a (primitive) function with result type *T* (a *function with a controlling result*), then the call has a *controlling result* — the context of the call can control the dispatching. Similarly, if the call is to a function with an access result type designating *T* (a *function with a controlling access result*), then the call has a *controlling access result*, and the context can similarly control dispatching.

Replace paragraph 20: [AI05-0126-1]

For the execution of a call on a dispatching operation, the action performed is determined by the properties of the corresponding dispatching operation of the specific type identified by the controlling tag value. If the corresponding operation is explicitly declared for this type, even if the declaration occurs in a private part, then the action comprises an invocation of the explicit body for the operation. If the corresponding operation is implicitly declared for this type:

by:

For the execution of a call on a dispatching operation, the action performed is determined by the properties of the corresponding dispatching operation of the specific type identified by the controlling tag value:

- if the corresponding operation is explicitly declared for this type, Redundant:[even if the declaration occurs in a private part], then the action comprises an invocation of the explicit body for the operation;

Replace paragraph 20.1: [AI05-0126-1]

- if the operation is implemented by an entry or protected subprogram (see 9.1 and 9.4), then the action comprises a call on this entry or protected subprogram, with the target object being given by the first actual parameter of the call, and the actual parameters of the entry or protected subprogram being given by the remaining actual parameters of the call, if any;

by:

- if the corresponding operation is implicitly declared for this type and is implemented by an entry or protected subprogram (see 9.1 and 9.4), then the action comprises a call on this entry or protected subprogram, with the target object being given by the first actual parameter of the call, and the actual parameters of the entry or protected subprogram being given by the remaining actual parameters of the call, if any;

Replace paragraph 20.2: [AI05-0126-1]

- otherwise, the action is the same as the action for the corresponding operation of the parent type.

by:

- otherwise, the action is the same as the action for the corresponding operation of the parent type or progenitor type from which the operation is inherited.

3.9.3 Abstract Types and Subprograms

Replace paragraph 4: [AI05-0097-1]

If a type has an implicitly declared primitive subprogram that is inherited or is the predefined equality operator, and the corresponding primitive subprogram of the parent or ancestor type is abstract or is a function with a controlling access result, or if a type other than a null extension inherits a function with a controlling result, then:

by:

If a type has an implicitly declared primitive subprogram that is inherited or is the predefined equality operator, and the corresponding primitive subprogram of the parent or ancestor type is abstract or is a function with a controlling access result, or if a type other than a non-abstract null extension inherits a function with a controlling result, then:

Replace paragraph 8: [AI05-0073-1]

The type of an aggregate, or of an object created by an object_declaration or an allocator, or a generic formal object of mode **in**, shall not be abstract. The type of the target of an assignment operation (see 5.2) shall not be abstract. The type of a component shall not be abstract. If the result type of a function is abstract, then the function shall be abstract.

by:

The type of an aggregate, or of an object created by an object_declaration or an allocator, or a generic formal object of mode **in**, shall not be abstract. The type of the target of an assignment operation (see 5.2) shall not be abstract. The type of a component shall not be abstract. If the result type of a function is abstract, then the function shall be abstract. If a function has an access result type designating an abstract type, then the function shall be abstract. A generic function shall not have an abstract result type or an access result type designating an abstract type.

Replace paragraph 10: [AI05-0073-1]

For an abstract type declared in a visible part, an abstract primitive subprogram shall not be declared in the private part, unless it is overriding an abstract subprogram implicitly declared in the visible part. For a tagged type declared in a visible part, a primitive function with a controlling result shall not be declared in the private part, unless it is overriding a function implicitly declared in the visible part.

by:

For an abstract type declared in a visible part, an abstract primitive subprogram shall not be declared in the private part, unless it is overriding an abstract subprogram implicitly declared in the visible part. For a tagged type declared in a visible part, a primitive function with a controlling result or a controlling access result shall not be declared in the private part, unless it is overriding a function implicitly declared in the visible part.

3.9.4 Interface Types

Replace paragraph 18: [AI05-0070-1]

The elaboration of an interface_type_definition has no effect.

by:

The elaboration of an interface_type_definition creates the interface type and its first subtype.

Replace paragraph 22: [AI05-0004-1]

Queue_Error : **exception**; -- *Append raises Queue_Error if Count(Q) = Max_Count(Q) -- Remove_First raises Queue_Error if Count(Q) = 0*

by:

Queue_Error : **exception**; -- *Append raises Queue_Error if Cur_Count(Q) = Max_Count(Q) -- Remove_First raises Queue_Error if Cur_Count(Q) = 0*

Replace paragraph 29: [AI05-0004-1]

```
type Fast_Food_Queue is new Queue with record ...;
procedure Append(Q : in out Fast_Food_Queue; Person : in Person_Name);
procedure Remove_First(Q : in out Fast_Food_Queue; Person : in Person_Name);
function Cur_Count(Q : in Fast_Food_Queue) return Natural;
function Max_Count(Q : in Fast_Food_Queue) return Natural;
```

by:

```
type Fast_Food_Queue is new Queue with record ...;
procedure Append(Q : in out Fast_Food_Queue; Person : in Person_Name);
procedure Remove_First(Q : in out Fast_Food_Queue; Person : out Person_Name);
function Cur_Count(Q : in Fast_Food_Queue) return Natural;
function Max_Count(Q : in Fast_Food_Queue) return Natural;
```

3.10 Access Types

Replace paragraph 9: [AI05-0053-1]

A view of an object is defined to be *aliased* if it is defined by an object_declaration or component_definition with the reserved word **aliased**, or by a renaming of an aliased view. In addition, the dereference of an access-to-object value denotes an aliased view, as does a view conversion (see 4.6) of an aliased view. The current instance of a limited tagged type, a protected type, a task type, or a type that has the reserved word **limited** in its full definition is also defined to be aliased. Finally, a formal parameter or generic formal object of a tagged type is defined to be aliased. Aliased views are the ones that can be designated by an access value.

by:

A view of an object is defined to be *aliased* if it is defined by an object_declaration or component_definition with the reserved word **aliased**, or by a renaming of an aliased view. In addition, the dereference of an access-to-object value denotes an aliased view, as does a view conversion (see 4.6) of an aliased view. The current instance of an immutably limited type (see 7.5) is defined to be aliased, as is the return object of an extended_return_statement (see 6.5) that is of an immutably limited type. Finally, a formal parameter or generic formal object of a tagged type is defined to be aliased. Aliased views are the ones that can be designated by an access value.

3.10.1 Incomplete Type Declarations

Replace paragraph 6: [AI05-0098-1]

- as the subtype_mark in the subtype_indication of an access_to_object_definition; the only form of constraint allowed in this subtype_indication is a discriminant_constraint;

by:

- as the subtype_mark in the subtype_indication of an access_to_object_definition; the only form of constraint

allowed in this subtype_indication is a discriminant_constraint (a null_exclusion is not allowed);

Replace paragraph 9.2: [AI05-0098-1]

- as the subtype_mark defining the subtype of a parameter or result of an access_to_subprogram_definition.

by:

- as the subtype_mark defining the subtype of a parameter or result of an access_to_subprogram_definition or an access_definition for an access-to-subprogram type.

3.10.2 Operations of Access Types

Replace paragraph 12.2: [AI05-0004-1]

- If the value of the access discriminant is determined by a component_association in an aggregate, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null);

by:

- If the value of the access discriminant is determined by a record_component_association in an aggregate, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null);

Replace paragraph 14.1: [AI05-0024-1]

- for an allocator used to define the constraint in a subtype_declaration, the level of the subtype_declaration;

by:

- for an allocator used to define the discriminant of an object, the level of the object;

Replace paragraph 14.2: [AI05-0024-1]

- for an allocator used to define the constraint in a component_definition, the level of the enclosing type;

by:

- for an allocator used to define the constraint in a subtype_indication in any other context, the level of the master that elaborates the subtype_indication.

Delete paragraph 14.3: [AI05-0024-1]

- for an allocator used to define the discriminant of an object, the level of the object.

Replace paragraph 14.4: [AI05-0024-1]

In this last case, the allocated object is said to be a *coextension* of the object whose discriminant designates it, as well as of any object of which the discriminated object is itself a coextension or subcomponent. All coextensions of an object are finalized when the object is finalized (see 7.6.1).

by:

In the first case, the allocated object is said to be a *coextension* of the object whose discriminant designates it, as well as of any object of which the discriminated object is itself a coextension or subcomponent. If the allocated object is a coextension of an anonymous object representing the result of an aggregate or function call that is used (in its entirety) to directly initialize a part of an object, after the result is assigned, the coextension becomes a coextension of the object being initialized and is no longer considered a coextension of the anonymous object. All coextensions of an object (which have not thus been transfered by such an initialization) are finalized when the object is finalized (see 7.6.1).

Replace paragraph 15: [AI05-0014-1]

- The accessibility level of a view of an object or subprogram denoted by a dereference of an access value is the same as that of the access type.

by:

- The accessibility level of a view of an object or subprogram designated by an access value is the same as that of the access type.

Replace paragraph 19: [AI05-0082-1]

The statically deeper relationship does not apply to the accessibility level of the anonymous type of an access parameter specifying an access-to-object type; that is, such an accessibility level is not considered to be statically deeper, nor statically shallower, than any other.

by:

The statically deeper relationship does not apply to the accessibility level of the anonymous type of an access parameter

specifying an access-to-object type nor does it apply to a descendant of a generic formal type; that is, such an accessibility level is not considered to be statically deeper, nor statically shallower, than any other.

Replace paragraph 26: [AI05-0008-1; AI05-0041-1]

- The view shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is constrained by its initial value.

by:

- The view shall not be a subcomponent that depends on discriminants of an object unless the object is known to be constrained.

Replace paragraph 27.2: [AI05-0041-1]

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

by:

- *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.

Replace paragraph 28: [AI05-0041-1]

- The accessibility level of the view shall not be statically deeper than that of the access type *A*. 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 view shall not be statically deeper than that of the access type *A*.

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

Section 4: Names and Expressions

4.1 Names

Replace paragraph 2: [AI05-0003-1]

name ::=
direct_name | explicit_dereference
| indexed_component | slice
| selected_component | attribute_reference
| type_conversion | function_call
| character_literal

by:

name ::=
direct_name | explicit_dereference
| indexed_component | slice
| selected_component | attribute_reference
| type_conversion | function_call
| character_literal | qualified_expression

Replace paragraph 7: [AI05-0004-1]

Certain forms of name (indexed_components, selected_components, slices, and attributes) include a prefix that is either itself a name that denotes some related entity, or an implicit_dereference of an access value that designates some related entity.

by:

Certain forms of name (indexed_components, selected_components, slices, and attribute_references) include a prefix that is either itself a name that denotes some related entity, or an implicit_dereference of an access value that designates some related entity.

Replace paragraph 9: [AI05-0008-1]

If the type of the name in a dereference is some access-to-object type *T*, then the dereference denotes a view of an object, the *nominal subtype* of the view being the designated subtype of *T*.

by:

If the type of the name in a dereference is some access-to-object type *T*, then the dereference denotes a view of an object, the *nominal subtype* of the view being the designated subtype of *T*. If the designated subtype has unconstrained discriminants, the (actual) subtype of the view is constrained by the values of the discriminants of the designated object, except when there is a partial view of the type of the designated subtype that does not have discriminants, in which case the dereference is not constrained by its discriminant values.

4.1.3 Selected Components

Replace paragraph 9.2: [AI05-0090-1]

The prefix (after any implicit dereference) shall resolve to denote an object or value of a specific tagged type *T* or class-wide type *T*'Class. The selector_name shall resolve to denote a view of a subprogram declared immediately within the declarative region in which an ancestor of the type *T* is declared. The first formal parameter of the subprogram shall be of type *T*, or a class-wide type that covers *T*, or an access parameter designating one of these types. The designator of the subprogram shall not be the same as that of a component of the tagged type visible at the point of the selected_component. The selected_component denotes a view of this subprogram that omits the first formal parameter. This view is called a *prefixed view* of the subprogram, and the prefix of the selected_component (after any implicit dereference) is called the *prefix* of the prefixed view.

by:

The prefix (after any implicit dereference) shall resolve to denote an object or value of a specific tagged type *T* or class-wide type *T*'Class. The selector_name shall resolve to denote a view of a subprogram declared immediately within the declarative region in which an ancestor of the type *T* is declared. The first formal parameter of the subprogram shall be of type *T*, or a class-wide type that covers *T*, or an access parameter designating one of these types. The designator of the subprogram shall not be the same as that of a component of the tagged type visible at the point of the selected_component. The subprogram shall not be an implicitly declared primitive operation of type *T* that overrides an inherited subprogram implemented by an entry or protected subprogram visible at the point of the selected_component. The selected_component denotes a view of this subprogram that omits the first formal

parameter. This view is called a *prefixed view* of the subprogram, and the prefix of the selected_component (after any implicit dereference) is called the *prefix* of the prefixed view.

4.1.4 Attributes

Replace paragraph 3: [AI05-0004-1]

attribute_designator ::=
identifier[(static_expression)]
| Access | Delta | Digits | Mod

by:

attribute_designator ::=
identifier[(static_expression)]
| Access | Delta | Digits

Replace paragraph 9: [AI05-0006-1]

An attribute_reference denotes a value, an object, a subprogram, or some other kind of program entity.

by:

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.

4.3.1 Record Aggregates

Replace paragraph 15: [AI05-0016-1]

If there are no components needed in a given record_component_association_list, then the reserved words **null record** shall appear rather than a list of record_component_associations.

by:

The reserved words **null record** may appear only if there are no components needed in a given record_component_association_list.

4.3.2 Extension Aggregates

Insert after paragraph 5: [AI05-0067-1]

If the ancestor_part is a subtype_mark, it shall denote a specific tagged subtype. If the ancestor_part is an expression, it shall not be dynamically tagged. The type of the extension_aggregate shall be derived from the type of the ancestor_part, through one or more record extensions (and no private extensions).

the new paragraph:

If the ancestor_part is a function call and the type of the ancestor_part is limited, then the ancestor_part shall have a constrained nominal subtype unless there are no components needed in the record_component_association_list.

4.3.3 Array Aggregates

Replace paragraph 29: [AI05-0037-1]

For an array_aggregate with an **others** choice, a check is made that no expression is specified for an index value outside the bounds determined by the applicable index constraint.

by:

For an array_aggregate with an **others** choice, a check is made that no expression or <> is specified for an index value outside the bounds determined by the applicable index constraint.

Replace paragraph 32: [AI05-0004-1]

10 In an array_aggregate, positional notation may only be used with two or more expressions; a single expression in parentheses is interpreted as a parenthesized_expression. A named_array_aggregate, such as (1 = X), may be used to specify an array with a single

component.>

by:

10 In an array_aggregate, positional notation may only be used with two or more expressions; a single expression in parentheses is interpreted as a parenthesized expression. A named_array_aggregate, such as (1 = X), may be used to specify an array with a single component.>

4.4 Expressions

Replace paragraph 7: [AI05-0003-1]

```
primary ::=
    numeric_literal | null | string_literal | aggregate
  | name | qualified_expression | allocator | (expression)
```

by:

```
primary ::=
    numeric_literal | null | string_literal | aggregate
  | name | allocator | (expression)
```

4.5.2 Relational Operators and Membership Tests

Replace paragraph 9.3: [AI05-0020-1]

- it is declared immediately within the same declaration list as *D*; and

by:

- it is declared immediately within the same declaration list as *D* or any partial or incomplete view of *D*; and

4.5.5 Multiplying Operators

Replace paragraph 19.2: [AI05-0020-1]

- it is declared immediately within the same declaration list as the type; and

by:

- it is declared immediately within the same declaration list as the type or any partial view thereof; and

4.5.6 Highest Precedence Operators

Replace paragraph 11: [AI05-0088-1]

The right operand of an exponentiation is the *exponent*. The expression X**N with the value of the exponent N positive is equivalent to the expression X*X*...X (with N–1 multiplications) except that the multiplications are associated in an arbitrary order. With N equal to zero, the result is one. With the value of N negative (only defined for a floating point operand), the result is the reciprocal of the result using the absolute value of N as the exponent.

by:

The right operand of an exponentiation is the *exponent*. The value of X**N with the value of the exponent N positive is the same as the value of X*X*...X (with N–1 multiplications) except that the multiplications are associated in an arbitrary order. With N equal to zero, the result is one. With the value of N negative (only defined for a floating point operand), the result is the reciprocal of the result using the absolute value of N as the exponent.

4.7 Qualified Expressions

Insert after paragraph 3: [AI05-0003-1]

The *operand* (the expression or aggregate) shall resolve to be of the type determined by the subtype_mark, or a universal type that covers it.

Insert after paragraph 3: [AI05-0003-1]

Static Semantics

If the operand of a qualified_expression denotes an object, the qualified_expression denotes a constant view of that object. The nominal subtype of a qualified_expression is the subtype denoted by the subtype_mark.

4.8 Allocators

Insert after paragraph 2: [AI05-0104-1]

allocator ::=
new subtype_indication | **new** qualified_expression

the new paragraph:

For an allocator with a subtype_indication, the subtype_indication shall not specify a null_exclusion.

Replace paragraph 5.3: [AI05-0052-1]

An allocator shall not be of an access type for which the Storage_Size has been specified by a static expression with value zero or is defined by the language to be zero. 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. This rule does not apply in the body of a generic unit or within a body declared within the declarative region of a generic unit, if the type of the allocator is a descendant of a formal access type declared within the formal part of the generic unit.

by:

An allocator shall not be of an access type for which the Storage_Size has been specified by a static expression with value zero or is defined by the language to be zero. This rule does not apply in the body of a generic unit or within a body declared within the declarative region of a generic unit, if the type of the allocator is a descendant of a formal access type declared within the formal part of the generic unit.

If the designated type of the type of the allocator is limited, then the allocator shall not be used to define the value of an access discriminant, unless the discriminated type is immutably limited (see 7.5).

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 6: [AI05-0041-1]

If the designated type of the type of the allocator is elementary, then the subtype of the created object is the designated subtype. If the designated type is composite, then the subtype of the created object is the designated subtype when the designated subtype is constrained or there is a partial view of the designated type that is constrained; otherwise, the created object is constrained by its initial value (even if the designated subtype is unconstrained with defaults).

by:

If the designated type of the type of the allocator is elementary, then the subtype of the created object is the designated subtype. If the designated type is composite, then the subtype of the created object is the designated subtype when the designated subtype is constrained or there is an ancestor of the designated type that has a constrained partial view; otherwise, the created object is constrained by its initial value (even if the designated subtype is unconstrained with defaults).

Replace paragraph 10.1: [AI05-0024-1]

For any allocator, if the designated type of the type of the allocator is class-wide, then a check is made that the accessibility level of the type determined by the subtype_indication, or by the tag of the value of the qualified_expression, is not deeper than that of the type of the allocator. If the designated subtype of the allocator has one or more unconstrained access discriminants, then a check is made that the accessibility level of the anonymous access type of each access discriminant is not deeper than that of the type of the allocator. Program_Error is raised if either such check fails.

by:

For any allocator, if the designated type of the type of the allocator is class-wide, then a check is made that the master of the type determined by the subtype_indication, or by the tag of the value of the qualified_expression, includes the elaboration of the type of the allocator. If the designated subtype of the allocator has one or more unconstrained access discriminants, then a check is made that the accessibility level of the anonymous access type of each access discriminant is not deeper than that of the type of the allocator. Program_Error is raised if either such check fails.

4.9.1 Statically Matching Constraints and Subtypes

Replace paragraph 4: [AI05-0086-1]

A constraint is *statically compatible* with a scalar subtype if it statically matches the constraint of the subtype, or if both are static and the constraint is compatible with the subtype. A constraint is *statically compatible* with an access or composite subtype if it statically matches the constraint of the subtype, or if the subtype is unconstrained. One subtype is *statically compatible* with a second subtype if the constraint of the first is statically compatible with the second subtype.

by:

A constraint is *statically compatible* with a scalar subtype if it statically matches the constraint of the subtype, or if both are static

and the constraint is compatible with the subtype. A constraint is *statically compatible* with an access or composite subtype if it statically matches the constraint of the subtype, or if the subtype is unconstrained. One subtype is *statically compatible* with a second subtype if the constraint of the first is statically compatible with the second subtype, and in the case of an access type, if the second subtype excludes null, then so does the first.

Section 5: Statements

5.4 Case Statements

Replace paragraph 7: [AI05-0003-1]

- If the expression is a name (including a type_conversion or a function_call) having a static and constrained nominal subtype, or is a qualified_expression whose subtype_mark denotes a static and constrained scalar subtype, then each non-others discrete_choice shall cover only values in that subtype, and each value of that subtype shall be covered by some discrete_choice (either explicitly or by others).

by:

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

Section 6: Subprograms

6.2 Formal Parameter Modes

Replace paragraph 7: [AI05-0096-1]

- [a nonprivate type with the reserved word @key[limited] in its declaration;

by:

- an explicitly limited record type;

6.3.1 Conformance Rules

Replace paragraph 18: [AI05-0046-1]

Two profiles are *fully conformant* if they are subtype-conformant, and corresponding parameters have the same names and have default_expressions that are fully conformant with one another.

by:

Two profiles are *fully conformant* if they are subtype-conformant, and corresponding parameters have the same names and both or neither have null_exclusions and they have default_expressions that are fully conformant with one another.

6.4.1 Parameter Associations

Replace paragraph 2: [AI05-0118-1]

The *formal_parameter_*selector_name of a parameter_association shall resolve to denote a parameter_specification of the view being called.

by:

The *formal_parameter_*selector_name of a {named} parameter_association shall resolve to denote a parameter_specification of the view being called{; this is the formal parameter of the association. The formal parameter for a positional parameter_association is the parameter with the corresponding position in the formal part of the view being called}.

Insert after paragraph 17: [AI05-0008-1]

After normal completion and leaving of a subprogram, for each **in out** or **out** parameter that is passed by copy, the value of the formal parameter is converted to the subtype of the variable given as the actual parameter and assigned to it. These conversions and assignments occur in an arbitrary order.

the new paragraph:

Erroneous Execution

If the nominal subtype of a formal parameter with discriminants is constrained or indefinite, and the parameter is passed by reference, then the execution of the call is erroneous if the value of any discriminant of the actual is changed while the formal parameter exists (that is, before leaving the corresponding callable construct).

6.5 Return Statements

Replace paragraph 2.1: [AI05-0015-1; AI05-0053-1]

```
extended_return_statement ::=
   return defining_identifier : [aliased] return_subtype_indication [:= expression] [do
      handled_sequence_of_statements
   end return];
```

by:

```
extended_return_statement ::=
   return defining_identifier : [constant] return_subtype_indication [:= expression] [do
      handled_sequence_of_statements
   end return];
```

Replace paragraph 5: [AI05-0015-1]

A function body shall contain at least one return statement that applies to the function body, unless the function contains code_statements. A simple_return_statement shall include an expression if and only if it applies to a function body. An

extended_return_statement shall apply to a function body.

by:

A function body shall contain at least one return statement that applies to the function body, unless the function contains code_statements. A simple_return_statement shall include an expression if and only if it applies to a function body. An extended_return_statement shall apply to a function body. An extended_return_statement with the reserved word **constant** shall include an expression.

Replace paragraph 5.2: [AI05-0032-1]

- If the result subtype of the function is defined by a subtype_mark, the return_subtype_indication shall be a subtype_indication. The type of the subtype_indication shall be the result type of the function. If the result subtype of the function is constrained, then the subtype defined by the subtype_indication shall also be constrained and shall statically match this result subtype. If the result subtype of the function is unconstrained, then the subtype defined by the subtype_indication shall be a definite subtype, or there shall be an expression.

by:

- If the result subtype of the function is defined by a subtype_mark, the return_subtype_indication shall be a subtype_indication. The type of the subtype_indication shall be covered by the result type of the function. If the result subtype of the function is constrained, then the subtype defined by the subtype_indication shall also be constrained and shall statically match this result subtype. If the result subtype of the function is unconstrained, then the subtype defined by the subtype_indication shall be a definite subtype, or there shall be an expression.

Insert after paragraph 5.3: [AI05-0032-1]

- If the result subtype of the function is defined by an access_definition, the return_subtype_indication shall be an access_definition. The subtype defined by the access_definition shall statically match the result subtype of the function. The accessibility level of this anonymous access subtype is that of the result subtype.

the new paragraph:

- If the result subtype of the function is class-wide, the accessibility level of the type of the subtype defined by the return_subtype_indication shall not be statically deeper than that of the master that elaborated the function body.

Replace paragraph 5.6: [AI05-0032-1]

- If the result subtype of the function is class-wide, the accessibility level of the type of the expression of the return statement shall not be statically deeper than that of the master that elaborated the function body. 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 simple_return_statement or the return_subtype_indication, shall not be statically deeper than that of the master that elaborated the function body.

by:

- If the result subtype of the function is class-wide, the accessibility level of the type of the expression (if any) of the return statement shall not be statically deeper than that of the master that elaborated the function body. 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 simple_return_statement or the return_subtype_indication, shall not be statically deeper than that of the master that elaborated the function body.

Replace paragraph 5.7: [AI05-0015-1]

Within an extended_return_statement, the *return object* is declared with the given defining_identifier, with the nominal subtype defined by the return_subtype_indication.

by:

Within an extended_return_statement, the *return object* is declared with the given defining_identifier, with the nominal subtype defined by the return_subtype_indication. An extended_return_statement with the reserved word **constant** is a full constant declaration for the return object.

Replace paragraph 5.8: [AI05-0032-1]

For the execution of an extended_return_statement, the subtype_indication or access_definition is elaborated. This creates the nominal subtype of the return object. If there is an expression, it is evaluated and converted to the nominal subtype (which might raise Constraint_Error — see 4.6); the return object is created and the converted value is assigned to the return object. Otherwise, the return object is created and initialized by default as for a stand-alone object of its nominal subtype (see 3.3.1). If the nominal subtype is indefinite, the return object is constrained by its initial value.

by:

For the execution of an extended_return_statement, the subtype_indication or access_definition is elaborated. This creates

the nominal subtype of the return object. If there is an expression, it is evaluated and converted to the nominal subtype (which might raise Constraint_Error — see 4.6); the return object is created and the converted value is assigned to the return object. Otherwise, the return object is created and initialized by default as for a stand-alone object of its nominal subtype (see 3.3.1). If the nominal subtype is indefinite, the return object is constrained by its initial value. A check is made that the value of the return object belongs to the function result subtype. Constraint_Error is raised if this check fails.

Replace paragraph 8: [AI05-0024-1; AI05-0032-1; AI05-0073-1]

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. A check is made that the accessibility level of the type identified by the tag of the result is not deeper than that 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 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.

If the result subtype of the function is defined by an access_definition designating a specific tagged type *T*, a check is made that the result value is null or the tag of the object designated by the result value identifies *T*. Constraint_Error is raised if this check fails.

Replace paragraph 22: [AI05-0058-1]

For the execution of an extended_return_statement, the handled_sequence_of_statements is executed. Within this handled_sequence_of_statements, the execution of a simple_return_statement that applies to the extended_return_statement causes a transfer of control that completes the extended_return_statement. Upon completion of a return statement that applies to a callable construct, a transfer of control is performed which completes the execution of the callable construct, and returns to the caller.

by:

For the execution of an extended_return_statement, the handled_sequence_of_statements is executed. Within this handled_sequence_of_statements, the execution of a simple_return_statement that applies to the extended_return_statement causes a transfer of control that completes the extended_return_statement. Upon completion of a return statement that applies to a callable construct by the normal completion of a simple_return_statement or by reaching the **end return** of an extended_return_statement, a transfer of control is performed which completes the execution of the callable construct, and returns to the caller.

Replace paragraph 24: [AI05-0050-1]

If the result subtype of a function is unconstrained, and a call on the function is used to provide the initial value of an object with a constrained nominal subtype, Constraint_Error may be raised at the point of the call (after abandoning the execution of the function body) if, while elaborating the return_subtype_indication or evaluating the expression of a return statement that applies to the function body, it is determined that the value of the result will violate the constraint of the subtype of this object.

by:

For a function call used to initialize a composite object with a constrained nominal subtype or used to initialize a return object that is built in place into such an object:

- If the result subtype of the function is constrained, and conversion of an object of this subtype to the subtype of the object being initialized would raise Constraint_Error, then Constraint_Error may be raised before calling the function.
- If the result subtype of the function is unconstrained, and a return statement is executed such that the return object is known to be constrained, and conversion of the return object to the subtype of the object being initialized would raise Constraint_Error, then Constraint_Error may be raised at the point of the call (after abandoning the execution of the function body).

Section 7: Packages

7.3.1 Private Operations

Replace paragraph 3: [AI05-0029-1]

For a composite type, the characteristics (see 7.3) of the type are determined in part by the characteristics of its component types. At the place where the composite type is declared, the only characteristics of component types used are those characteristics visible at that place. If later immediately within the declarative region in which the composite type is declared additional characteristics become visible for a component type, then any corresponding characteristics become visible for the composite type. Any additional predefined operators are implicitly declared at that place.

by:

For a composite type, the characteristics (see 7.3) of the type are determined in part by the characteristics of its component types. At the place where the composite type is declared, the only characteristics of component types used are those characteristics visible at that place. If later immediately within the declarative region in which the composite type is declared additional characteristics become visible for a component type, then any corresponding characteristics become visible for the composite type. Any additional predefined operators are implicitly declared at that place. If there is no such place, then additional predefined operators are not declared at all, but they still exist.

Replace paragraph 6: [AI05-0029-1]

Inherited primitive subprograms follow a different rule. For a derived_type_definition, each inherited primitive subprogram is implicitly declared at the earliest place, if any, immediately within the declarative region in which the type_declaration occurs, but after the type_declaration, where the corresponding declaration from the parent is visible. If there is no such place, then the inherited subprogram is not declared at all. An inherited subprogram that is not declared at all cannot be named in a call and cannot be overridden, but for a tagged type, it is possible to dispatch to it.

by:

Inherited primitive subprograms follow a different rule. For a derived_type_definition, each inherited primitive subprogram is implicitly declared at the earliest place, if any, immediately within the declarative region in which the type_declaration occurs, but after the type_declaration, where the corresponding declaration from the parent is visible. If there is no such place, then the inherited subprogram is not declared at all, but it still exists. For a tagged type, it is possible to dispatch to an inherited subprogram that is not declared at all.

Replace paragraph 10.1: [AI05-0004-1]

If the full_type_declaration for a private extension is a derived_type_declaration, then the reserved word **limited** shall appear in the full_type_declaration if and only if it also appears in the private_extension_declaration.

by:

If the full_type_declaration for a private extension is defined by a derived_type_definition, then the reserved word **limited** shall appear in the full_type_declaration if and only if it also appears in the private_extension_declaration.

7.4 Deferred Constants

Replace paragraph 6: [AI05-0062-1]

- If the deferred constant declaration includes a subtype_indication that defines a constrained subtype, then the subtype defined by the subtype_indication in the full declaration shall match it statically. On the other hand, if the subtype of the deferred constant is unconstrained, then the full declaration is still allowed to impose a constraint. The constant itself will be constrained, like all constants;

by:

- If the deferred constant declaration includes a subtype_indication that defines a constrained subtype, then the constraint defined by the subtype_indication in the full declaration shall match it statically. On the other hand, if the subtype of the deferred constant is unconstrained, then the full declaration is still allowed to impose a constraint. The constant itself will be constrained, like all constants;

Replace paragraph 10: [AI05-0004-1]

The elaboration of a deferred constant declaration elaborates the subtype_indication or (only allowed in the case of an imported constant) the array_type_definition.

by:

The elaboration of a deferred constant declaration elaborates the subtype_indication, access_definition, or (only allowed in the

case of an imported constant) the array_type_definition.

7.5 Limited Types

Insert after paragraph 4: [AI05-0087-1]

- a type with the reserved word **limited**, **synchronized**, **task**, or **protected** in its definition;

the new paragraph:

- a class-wide type whose specific type is limited;

Insert after paragraph 8: [AI05-0052-1]

There are no predefined equality operators for a limited type.

the new paragraphs:

A type is *immutably limited* if it is one of the following:

- An explicitly limited record type;
- A non-formal limited private type that is tagged or has at least one access discriminant with a default_expression;
- A task type, a protected type, or a synchronized interface;
- A descendant of an immutably limited type.

A descendant of a generic formal limited private type is presumed to be immutably limited except within the body of a generic unit or a body declared within the declarative region of a generic unit, if the formal type is declared within the formal part of the generic unit.

Delete paragraph 8.1: [AI05-0067-1]

Implementation Requirements

For an aggregate of a limited type used to initialize an object as allowed above, the implementation shall not create a separate anonymous object for the aggregate. For a function_call of a type with a part that is of a task, protected, or explicitly limited record type that is used to initialize an object as allowed above, the implementation shall not create a separate return object (see 6.5) for the function_call. The aggregate or function_call shall be constructed directly in the new object.

Replace paragraph 9: [AI05-0067-1]

> 14 While it is allowed to write initializations of limited objects, such initializations never copy a limited object. The source of such an assignment operation must be an aggregate or function_call, and such aggregates and function_calls must be built directly in the target object.

by:

> 14 While it is allowed to write initializations of limited objects, such initializations never copy a limited object. The source of such an assignment operation must be an aggregate or function_call, and such aggregates and function_calls must be built directly in the target object (see 7.6).

7.6 User-Defined Assignment and Finalization

Replace the title: [AI05-0067-1]

User-Defined Assignment and Finalization

by:

Assignment and Finalization

Replace paragraph 9.4: [AI05-0013-1; AI05-0026-1]

- it is a limited type that has an access discriminant whose designated type needs finalization; or

by:

- it is a class-wide type; or
- it is a partial view whose full view needs finalization; or

Replace paragraph 16: [AI05-0067-1]

To adjust the value of a (nonlimited) composite object, the values of the components of the object are first adjusted in an arbitrary order, and then, if the object is controlled, Adjust is called. Adjusting the value of an elementary object has no effect, nor does

adjusting the value of a composite object with no controlled parts.

by:

To adjust the value of a composite object, the values of the components of the object are first adjusted in an arbitrary order, and then, if the object is nonlimited controlled, Adjust is called. Adjusting the value of an elementary object has no effect, nor does adjusting the value of a composite object with no controlled parts.

Insert after paragraph 17: [AI05-0067-1]

For an assignment_statement, after the name and expression have been evaluated, and any conversion (including constraint checking) has been done, an anonymous object is created, and the value is assigned into it; that is, the assignment operation is applied. (Assignment includes value adjustment.) The target of the assignment_statement is then finalized. The value of the anonymous object is then assigned into the target of the assignment_statement. Finally, the anonymous object is finalized. As explained below, the implementation may eliminate the intermediate anonymous object, so this description subsumes the one given in 5.2, "Assignment Statements".

the new paragraphs:

When a function call or aggregate is used to initialize an object, the result of the function call or aggregate is an anonymous object, which is assigned into the newly-created object. For such an assignment, the anonymous object might be built in place. Under certain circumstances, the anonymous object is required to be *built in place*, in which case the assignment does not involve any copying. In particular:

- If the full type of any part of the object is immutably limited, the anonymous object is built in place.
- In the case of an aggregate, if the full type of any part of the newly-created object is controlled, the anonymous object is built in place.
- In other cases, it is unspecified whether the anonymous object is built in place.

Notwithstanding what this International Standard says elsewhere, if an object is built in place:

- Upon successful completion of the return statement or aggregate, the anonymous object *mutates into* the newly-created object; that is, the anonymous object ceases to exist, and the newly-created object appears in its place.
- Finalization is not performed on the anonymous object.
- Adjustment is not performed on the newly-created object.
- All access values that designate parts of the anonymous object now designate the corresponding parts of the newly-created object.
- All renamings of parts of the anonymous object now denote views of the corresponding parts of the newly-created object.
- Coextensions of the anonymous object become coextensions of the newly-created object.

Delete paragraph 17.1: [AI05-0067-1]

Implementation Requirements

For an aggregate of a controlled type whose value is assigned, other than by an assignment_statement, the implementation shall not create a separate anonymous object for the aggregate. The aggregate value shall be constructed directly in the target of the assignment operation and Adjust is not called on the target object.

Replace paragraph 18: [AI05-0067-1]

An implementation is allowed to relax the above rules (for nonlimited controlled types) in the following ways:

by:

An implementation is allowed to relax the above rules for assignment_statements in the following ways:

Replace paragraph 19: [AI05-0067-1]

For an assignment_statement that assigns to an object the value of that same object, the implementation need not do anything.

by:

If an object is assigned the value of that same object, the implementation need not do anything.

Replace paragraph 20: [AI05-0067-1]

For an assignment_statement for a noncontrolled type, the implementation may finalize and assign each component of the variable separately (rather than finalizing the entire variable and assigning the entire new value) unless a discriminant of the variable is changed by the assignment.

by:

For assignment of a noncontrolled type, the implementation may finalize and assign each component of the variable separately (rather than finalizing the entire variable and assigning the entire new value) unless a discriminant of the variable is changed by the assignment.

Replace paragraph 21: [AI05-0067-1]

- For an aggregate or function call whose value is assigned into a target object, the implementation need not create a separate anonymous object if it can safely create the value of the aggregate or function call directly in the target object. Similarly, for an assignment_statement, the implementation need not create an anonymous object if the value being assigned is the result of evaluating a name denoting an object (the source object) whose storage cannot overlap with the target. If the source object might overlap with the target object, then the implementation can avoid the need for an intermediary anonymous object by exercising one of the above permissions and perform the assignment one component at a time (for an overlapping array assignment), or not at all (for an assignment where the target and the source of the assignment are the same object).

by:

- The implementation need not create an anonymous object if the value being assigned is the result of evaluating a name denoting an object (the source object) whose storage cannot overlap with the target. If the source object might overlap with the target object, then the implementation can avoid the need for an intermediary anonymous object by exercising one of the above permissions and perform the assignment one component at a time (for an overlapping array assignment), or not at all (for an assignment where the target and the source of the assignment are the same object).

7.6.1 Completion and Finalization

Replace paragraph 6: [AI05-0099-1]

- If the object is of an elementary type, finalization has no effect;

by:

- If the full type of the object is an elementary type, finalization has no effect;

Replace paragraph 7: [AI05-0099-1]

- If the object is of a controlled type, the Finalize procedure is called;

by:

- If the full type of the object is a tagged type, and the tag of the object identifies a controlled type, the Finalize procedure of that controlled type is called;

Replace paragraph 8: [AI05-0099-1]

- If the object is of a protected type, the actions defined in 9.4 are performed;

by:

- If the full type of the object is a protected type, or if the full type of the object is a tagged type and the tag of the object identifies a protected type, the actions defined in 9.4 are performed;

Replace paragraph 9: [AI05-0099-1]

- If the object is of a composite type, then after performing the above actions, if any, every component of the object is finalized in an arbitrary order, except as follows: if the object has a component with an access discriminant constrained by a per-object expression, this component is finalized before any components that do not have such discriminants; for an object with several components with such a discriminant, they are finalized in the reverse of the order of their component_declarations;

by:

- If the full type of the object is a composite type, then after performing the above actions, if any, every component of the object is finalized in an arbitrary order, except as follows: if the object has a component with an access discriminant constrained by a per-object expression, this component is finalized before any components that do not have such discriminants; for an object with several components with such a discriminant, they are finalized in the reverse of the order of their component_declarations;

Replace paragraph 13: [AI05-0066-1]

The master of an object is the master enclosing its creation whose accessibility level (see 3.10.2) is equal to that of the object.

by:

The master of an object is the master enclosing its creation whose accessibility level (see 3.10.2) is equal to that of the object, except in the case of an anonymous object representing the result of an aggregate or function call. The master of such an anonymous object is the innermost master enclosing the evaluation of the aggregate or function call, which may be the aggregate or function call itself.

Replace paragraph 13.1: [AI05-0066-1]

In the case of an expression that is a master, finalization of any (anonymous) objects occurs as the final part of evaluation of the expression.

by:

In the case of an expression that is a master, finalization of any (anonymous) objects occurs after completing evaluation of the expression and use of the objects, prior to starting the execution of any subsequent construct.

Delete paragraph 17.1: [AI05-0064-1]

- For a Finalize invoked as part of the finalization of the anonymous object created by a function call or aggregate, any other finalizations due to be performed are performed, and then Program_Error is raised.

Section 8: Visibility Rules

8.5.1 Object Renaming Declarations

Replace paragraph 5: [AI05-0008-1]

The renamed entity shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is constrained by its initial value. A slice of an array shall not be renamed if this restriction disallows renaming of the array. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit. These rules also apply for a renaming that appears in the body of a generic unit, with the additional requirement that even if the nominal subtype of the variable is indefinite, its type shall not be a descendant of an untagged generic formal derived type.

by:

The renamed entity shall not be a subcomponent that depends on discriminants of an object whose nominal subtype is unconstrained unless the object is known to be constrained. A slice of an array shall not be renamed if this restriction disallows renaming of the array. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit.

Section 9: Tasks and Synchronization

9.1 Task Units and Task Objects

Replace paragraph 9.2: [AI05-0042-1]

For a task declaration with an interface_list, the task type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the task type or an access parameter designating the task type, and there is an entry_declaration for a single entry with the same identifier within the task declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming task entry.

by:

For a task declaration with an interface_list, the task type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the task type or an access parameter designating the task type, and there is an entry_declaration for a single entry with the same identifier within the task declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming task entry using an implicitly declared non-abstract subprogram which has the same profile as the inherited subprogram and which overrides it.

Replace paragraph 9.5: [AI05-0090-1]

The prefixed view profile of an explicitly declared primitive subprogram of a tagged task type shall not be type conformant with any entry of the task type, if the first parameter of the subprogram is of the task type or is an access parameter designating the task type.

by:

The prefixed view profile of an explicitly declared primitive subprogram of a tagged task type shall not be type conformant with any entry of the task type, if the subprogram has the same defining name as the entry and the first parameter of the subprogram is of the task type or is an access parameter designating the task type.

9.2 Task Execution - Task Activation

Replace paragraph 6: [AI05-0045-1]

Should the task that created the new tasks never reach the point where it would initiate the activations (due to an abort or the raising of an exception), the newly created tasks become terminated and are never activated.

by:

If the master that directly encloses the point where the activation of a task *T* would be initiated, completes before the activation of *T* is initiated, *T* becomes terminated and is never activated. Furthermore, if a return statement is left such that the return object is not returned to the caller, any task that was created as a part of the return object or one of its coextensions immediately becomes terminated and is never activated.

9.4 Protected Units and Protected Objects

Replace paragraph 11.1: [AI05-0042-1]

For a protected declaration with an interface_list, the protected type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the protected type or an access parameter designating the protected type, and there is a protected_operation_declaration for a protected subprogram or single entry with the same identifier within the protected declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming protected subprogram or entry.

by:

For a protected declaration with an interface_list, the protected type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the protected type or an access parameter designating the protected type, and there is a protected_operation_declaration for a protected subprogram or single entry with the same identifier within the protected declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming protected subprogram or entry using

an implicitly declared non-abstract subprogram which has the same profile as the inherited subprogram and which overrides it.

Replace paragraph 11.4: [AI05-0090-1]

The prefixed view profile of an explicitly declared primitive subprogram of a tagged protected type shall not be type conformant with any protected operation of the protected type, if the first parameter of the subprogram is of the protected type or is an access parameter designating the protected type.

by:

The prefixed view profile of an explicitly declared primitive subprogram of a tagged protected type shall not be type conformant with any protected operation of the protected type, if the subprogram has the same defining name as the protected operation and the first parameter of the subprogram is of the protected type or is an access parameter designating the protected type.

9.5 Intertask Communication

Insert after paragraph 9: [AI05-0030-2]

Any call on a protected procedure or entry of a target protected object is defined to be an update to the object, as is a requeue on such an entry.

the new paragraphs:

Syntax

The form of a pragma Implemented is as follows **pragma** Implemented(*procedure*_local_name, implementation_kind); implementation_kind ::= By_Entry | By_Protected_Procedure | By_Any

Legality Rules

The *procedure*_local_name of a pragma Implemented shall denote a primitive procedure of a synchronized tagged type.

A pragma Implemented with implementation_kind By_Protected shall not be applied to a primitive procedure of a task interface.

A procedure for which the implementation_kind is specified as By_Entry shall be implemented by an entry. A procedure for which the implementation_kind is specified as By_Protected_Procedure shall be implemented by a protected procedure.

If a primitive procedure overrides an inherited operation to which a pragma Implemented applies then any pragma Implemented applied to the overriding operation shall have the same implementation_kind or the inherited implementation_kind shall be By_Any.

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.

Static Semantics

A pragma Implemented is said to *apply* to the procedure denoted by its *procedure*_local_name. If an overriding operation does not have a pragma Implemented then any pragma Implemented applying to the inherited operation applies to the overriding operation.

NOTE 18 The implementation_kind By_Protected_Procedure implies that the operation will not block.

9.5.4 Requeue Statements

Replace paragraph 2: [AI05-0030-2]

requeue_statement ::= **requeue** entry_name [**with abort**];

by:

requeue_statement ::= **requeue** procedure_or_entry_name [**with abort**];

Replace paragraph 3: [AI05-0030-2]

The *entry*_name of a requeue_statement shall resolve to denote an entry (the *target entry*) that either has no parameters, or that has a profile that is type conformant (see 6.3.1) with the profile of the innermost enclosing entry_body or accept_statement.

by:

The *procedure_or_entry*_name of a requeue_statement shall resolve to denote a procedure or an entry (the *requeue target*) that either has no parameters, or that has a profile that is type conformant (see 6.3.1) with the profile of the innermost enclosing entry_body or accept_statement.

Replace paragraph 5: [AI05-0030-2]

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

by:

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

If the target is a procedure, the name shall denote a rename of an entry, or shall denote a prefixed view of a primitive subprogram of a synchronized interface, where the first parameter of the unprefixed view of the primitive subprogram shall be a controlling parameter, and a pragma Implemented with implementation_kind By_Entry shall apply to the primitive subprogram.

Replace paragraph 6: [AI05-0030-2]

In a requeue_statement of an accept_statement of some task unit, either the target object shall be a part of a formal parameter of the accept_statement, or the accessibility level of the target object shall not be equal to or statically deeper than any enclosing accept_statement of the task unit. In a requeue_statement of an entry_body of some protected unit, either the target object shall be a part of a formal parameter of the entry_body, or the accessibility level of the target object shall not be statically deeper than that of the entry_declaration.

by:

In a requeue_statement of an accept_statement of some task unit, either the target object shall be a part of a formal parameter of the accept_statement, or the accessibility level of the target object shall not be equal to or statically deeper than any enclosing accept_statement of the task unit. In a requeue_statement of an entry_body of some protected unit, either the target object shall be a part of a formal parameter of the entry_body, or the accessibility level of the target object shall not be statically deeper than that of the entry_declaration for the entry_body.

Replace paragraph 7: [AI05-0030-2]

The execution of a requeue_statement proceeds by first evaluating the *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. The entry_body or accept_statement enclosing the requeue_statement is then completed, finalized, and left (see 7.6.1).

Replace paragraph 12: [AI05-0030-2]

If the new entry 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, 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.

9.10 Shared Variables

Replace paragraph 6.1: [AI05-0072-1]

- If A1 is the termination of a task T, and A2 is either the evaluation of the expression T'Terminated or a call to Ada.Task_Identification.Is_Terminated with an actual parameter that identifies T (see C.7.1).

by:

- If A1 is the termination of a task T, and A2 is either an evaluation of the expression T'Terminated that results in True, or a call to Ada.Task_Identification.Is_Terminated with an actual parameter that identifies T and a result of True (see C.7.1).

Section 10: Program Structure and Compilation Issues

10.1.1 Compilation Units - Library Units

Replace paragraph 17: [AI05-0004-1]

A child of a generic library package shall either be itself a generic unit or be a renaming of some other child of the same generic unit. The renaming of a child of a generic package shall occur only within the declarative region of the generic package.

by:

A child of a generic library package shall either be itself a generic unit or be a renaming of some other child of the same generic unit.

10.1.2 Context Clauses - With Clauses

Replace paragraph 12: [AI05-0077-1]

A name denoting a library item that is visible only due to being mentioned in one or more with_clauses that include the reserved word **private** shall appear only within:

by:

A name denoting a library_item that is visible only due to being mentioned in one or more with_clauses that include the reserved word **private** shall appear only within:

Replace paragraph 12.3: [AI05-0108-1]

- For each type_declaration in the visible part, an incomplete view of the type; if the type_declaration is tagged, then the view is a tagged incomplete view.

by:

- For each type_declaration in the visible part, an incomplete view of the type with no discriminant_part; if the type_declaration is tagged, then the view is a tagged incomplete view.

Replace paragraph 20: [AI05-0040-1]

in the context_clause for the explicit declaration of the named library package;

by:

in the context_clause for the explicit declaration of the named library package or any of its descendants;

Replace paragraph 21: [AI05-0077-1]

- in the same context_clause as, or within the scope of, a nonlimited_with_clause that mentions the same library package; or

by:

- within a context_clause for a library_item which is within the scope of a nonlimited_with_clause that mentions the same library package; or

Replace paragraph 22: [AI05-0077-1]

- in the same context_clause as, or within the scope of, a use_clause that names an entity declared within the declarative region of the library package.

by:

- within a context_clause for a library_item which is within the scope of a use_clause that names an entity declared in the declarative region of the library package.

10.1.3 Subunits of Compilation Units

Replace paragraph 10: [AI05-0004-1]

A package_body_stub shall be the completion of a package_declaration or generic_package_declaration; a task_body_stub shall be the completion of a task_declaration; a protected_body_stub shall be the completion of a protected_declaration.

by:

A package_body_stub shall be the completion of a package_declaration or generic_package_declaration; a task_body_stub shall be the completion of a task declaration; a protected_body_stub shall be the completion of a protected declaration.

10.2.1 Elaboration Control

Replace paragraph 9: [AI05-0028-1]

- The creation of an object (including a component) of a type that does not have preelaborable initialization. Similarly, the evaluation of an extension_aggregate with an ancestor subtype_mark denoting a subtype of such a type.

by:

- The creation of an object (including a component) that is initialized by default, if its type does not have preelaborable initialization. Similarly, of a descendant of a private type, private extension, controlled type, task type, or protected type with entry_declarations; similarly the evaluation of an extension_aggregate with an ancestor subtype_mark denoting a subtype of such a type.

Replace paragraph 10.1: [AI05-0028-1]

- the actual for each formal private type (or extension) declared within the formal part of the generic unit is a private type (or extension) that does not have preelaborable initialization;

by:

- the actual for each discriminated formal derived type, formal private type, or formal private extension declared within the formal part of the generic unit is a type that does not have preelaborable initialization, unless pragma Preelaborable_Initialization has been applied to the formal type;

Replace paragraph 11: [AI05-0034-1]

If a pragma Preelaborate (or pragma Pure — see below) applies to a library unit, then it is *preelaborated*. If a library unit is preelaborated, then its declaration, if any, and body, if any, are elaborated prior to all non-preelaborated library_items of the partition. The declaration and body of a preelaborated library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be preelaborable. 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. In addition, all compilation units of a preelaborated library unit shall depend semantically only on compilation units of other preelaborated library units.

by:

If a pragma Preelaborate (or pragma Pure — see below) applies to a library unit, then it is *preelaborated*. The declaration and body of a preelaborated library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be preelaborable. In addition, the limited view of a library package is preelaborated. All compilation units of a preelaborated library unit shall depend semantically only on compilation units of other preelaborated library units. 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. If a library unit is preelaborated, then its declaration, if any, and body, if any, are elaborated prior to all non-preelaborated library_items of the partition.

Replace paragraph 11.2: [AI05-0028-1]

- The partial view of a private type or private extension, a protected type without entry_declarations, a generic formal private type, or a generic formal derived type, have preelaborable initialization if and only if the pragma Preelaborable_Initialization has been applied to them. A protected type with entry_declarations or a task type never has preelaborable initialization.

by:

- The partial view of a private type or private extension, a protected type without entry_declarations, a generic formal private type, or a generic formal derived type, has preelaborable initialization if and only if the pragma Preelaborable_Initialization has been applied to them. A protected type with entry_declarations or a task type never has preelaborable initialization.

Replace paragraph 11.4: [AI05-0028-1]

- A derived type has preelaborable initialization if its parent type has preelaborable initialization and (in the case of a derived record extension) if the non-inherited components all have preelaborable initialization. However, a user-defined controlled type with an overriding Initialize procedure does not have preelaborable initialization.

by:

- A derived type has preelaborable initialization if its parent type has preelaborable initialization and (in the case of a

derived record extension) if the non-inherited components all have preelaborable initialization. However, a controlled type with an Initialize procedure that is not a null procedure does not have preelaborable initialization.

Replace paragraph 11.7: [AI05-0028-1]

If the pragma appears in the first list of basic_declarative_items of a package_specification, then the direct_name shall denote the first subtype of a private type, private extension, or protected type that is not an interface type and is without entry_declarations, and the type shall be declared immediately within the same package as the pragma. If the pragma is applied to a private type or a private extension, the full view of the type shall have preelaborable initialization. If the pragma is applied to a protected type, each component of the protected type shall have preelaborable initialization. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit.

by:

If the pragma appears in the first list of basic_declarative_items of a package_specification, then the direct_name shall denote the first subtype of a composite type, and the type shall be declared immediately within the same package as the pragma. If the pragma is applied to a private type or a private extension, the full view of the type shall have preelaborable initialization. If the pragma is applied to a protected type, the protected type shall not have entries, and each component of the protected type shall have preelaborable initialization. For any other composite type, the type shall have preelaborable initialization. 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 15.1: [AI05-0035-1]

A *pure* library_item is a preelaborable library_item whose elaboration does not perform any of the following actions:

by:

A *pure* compilation unit is a preelaborable compilation unit whose elaboration does not perform any of the following actions:

Replace paragraph 15.4: [AI05-0035-1]

- the elaboration of the declaration of a named access-to-variable type unless the Storage_Size of the type has been specified by a static expression with value zero or is defined by the language to be zero;

by:

- the elaboration of the declaration of a non-derived named access-to-variable type unless the Storage_Size of the type has been specified by a static expression with value zero or is defined by the language to be zero;

Replace paragraph 15.5: [AI05-0035-1]

- the elaboration of the declaration of a named access-to-constant type for which the Storage_Size has been specified by an expression other than a static expression with value zero.

by:

- the elaboration of the declaration of a non-derived named access-to-constant type for which the Storage_Size has been specified by an expression other than a static expression with value zero.

A generic body is pure only if elaboration of a corresponding instance body would not perform any such actions presuming any composite formal types have non-visible components whose default initialization evaluates an allocator of an access-to-variable type.

Replace paragraph 17: [AI05-0034-1; AI05-0035-1]

A pragma Pure is used to declare that a library unit is pure. If a pragma Pure applies to a library unit, then its compilation units shall be pure, and they shall depend semantically only on compilation units of other library units that are declared pure. Furthermore, the full view of any partial view declared in the visible part of the library unit that has any available stream attributes shall support external streaming (see 13.13.2).

by:

A pragma Pure is used to declare that a library unit is 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. In addition, the limited view of a library package is declared pure. All compilation units of a declared pure library unit shall depend semantically only on compilation units of other library units that are declared pure. 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 the library unit that has any available stream attributes shall support external streaming (see 13.13.2).

Section 11: Exceptions

Replace paragraph 2: [AI05-0043-1]

An exception_declaration declares a name for an exception. An exception is raised initially either by a raise_statement or by the failure of a language-defined check. When an exception arises, control can be transferred to a user-provided exception_handler at the end of a handled_sequence_of_statements, or it can be propagated to a dynamically enclosing execution.

by:

An exception_declaration declares a name for an exception. An exception is raised initially by a raise_statement, by a call to Raise_Exception, or by the failure of a language-defined check. When an exception arises, control can be transferred to a user-provided exception_handler at the end of a handled_sequence_of_statements, or it can be propagated to a dynamically enclosing execution.

11.4.1 The Package Exceptions

Replace paragraph 10.1: [AI05-0043-1]

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. 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 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 for some other reason, the message is an unspecified string. In all cases, Exception_Message returns a string with lower bound 1.

Section 12: Generic Units

12.3 Generic Instantiation

Replace paragraph 7: [AI05-0004-1]

The *generic actual parameter* is either the explicit_generic_actual_parameter given in a generic_parameter_association for each formal, or the corresponding default_expression or default_name if no generic_parameter_association is given for the formal. When the meaning is clear from context, the term "generic actual," or simply "actual," is used as a synonym for "generic actual parameter" and also for the view denoted by one, or the value of one.

by:

The *generic actual parameter* is either the explicit_generic_actual_parameter given in a generic_association for each formal, or the corresponding default_expression or default_name if no generic_association is given for the formal. When the meaning is clear from context, the term "generic actual," or simply "actual," is used as a synonym for "generic actual parameter" and also for the view denoted by one, or the value of one.

Replace paragraph 9: [AI05-0118-1]

The *generic_formal_parameter_*selector_name of a generic_association shall denote a generic_formal_parameter_declaration of the generic unit being instantiated. If two or more formal subprograms have the same defining name, then named associations are not allowed for the corresponding actuals.

by:

The *generic_formal_parameter_*selector_name of a named generic_association shall denote a generic_formal_parameter_declaration of the generic unit being instantiated. If two or more formal subprograms have the same defining name, then named associations are not allowed for the corresponding actuals.

The generic_formal_parameter_declaration for a positional generic_association is the parameter with the corresponding position in the generic_formal_part of the generic unit being instantiated.

12.5 Formal Types

Replace paragraph 8: [AI05-0029-1]

The formal type also belongs to each category that contains the determined category. The primitive subprograms of the type are as for any type in the determined category. For a formal type other than a formal derived type, these are the predefined operators of the type. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later immediately within the declarative region in which the type is declared according to the rules of 7.3.1. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type. The rules specific to formal derived types are given in 12.5.1.

by:

The formal type also belongs to each category that contains the determined category. The primitive subprograms of the type are as for any type in the determined category. For a formal type other than a formal derived type, these are the predefined operators of the type. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later immediately within the declarative region in which the type is declared according to the rules of 7.3.1. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type and even if it is never declared for the actual type. The rules specific to formal derived types are given in 12.5.1.

12.5.1 Formal Private and Derived Types

Replace paragraph 5.1: [AI05-0087-1]

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 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. If the reserved word **synchronized** appears in the declaration of the formal derived type, the actual type shall be a synchronized tagged type.

Replace paragraph 21: [AI05-0029-1]

For a formal derived type, the predefined operators and inherited user-defined subprograms are determined by the ancestor type and any progenitor types, and are implicitly declared at the earliest place, if any, immediately within the declarative region in which the formal type is declared, where the corresponding primitive subprogram of the ancestor or progenitor is visible (see 7.3.1). In an instance, the copy of such an implicit declaration declares a view of the corresponding primitive subprogram of the ancestor or progenitor of the formal derived type, even if this primitive has been overridden for the actual type. When the ancestor or progenitor of the formal derived type is itself a formal type, the copy of the implicit declaration declares a view of the corresponding copied operation of the ancestor or progenitor. In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type.

by:

For a formal derived type, the predefined operators and inherited user-defined subprograms are determined by the ancestor type and any progenitor types, and are implicitly declared at the earliest place, if any, immediately within the declarative region in which the formal type is declared, where the corresponding primitive subprogram of the ancestor or progenitor is visible (see 7.3.1). In an instance, the copy of such an implicit declaration declares a view of the corresponding primitive subprogram of the ancestor or progenitor of the formal derived type, even if this primitive has been overridden for the actual type and even if it is never declared for the actual type. When the ancestor or progenitor of the formal derived type is itself a formal type, the copy of the implicit declaration declares a view of the corresponding copied operation of the ancestor or progenitor. In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type.

Replace paragraph 23.1: [AI05-0071-1]

In the case where a formal type is tagged with unknown discriminants, and the actual type is a class-wide type *T*'Class:

by:

In the case where a formal type has unknown discriminants, and the actual type is a class-wide type *T*'Class:

12.6 Formal Subprograms

Replace paragraph 10: [AI05-0071-1]

If a generic unit has a subprogram_default specified by a box, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a usage name identical to the defining name of the formal.

by:

If a generic unit has a subprogram_default specified by a box, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a usage name identical to the defining name of the formal. If a subtype_mark in the profile of the formal_subprogram_declaration denotes a formal private or formal derived type and the actual type for this formal type is a class-wide type *T*'Class, then for the purposes of resolving this default_name at the point of the instantiation, for each primitive subprogram of *T* that has a matching defining name, that is directly visible at the point of the instantiation, and that has at least one controlling formal parameter, a corresponding subprogram with the same defining name is directly visible, but with *T* systematically replaced by *T*'Class in the types of its profile. The body of such a subprogram is as defined in 12.5.1 for primitive subprograms of a formal type when the actual type is class-wide.

12.7 Formal Packages

Insert after paragraph 4: [AI05-0025-1]

The *generic_package_*name shall denote a generic package (the *template* for the formal package); the formal package is an instance of the template.

the new paragraph:

The *generic_formal_parameter_*selector_name of a formal_package_association shall denote a generic_formal_parameter_declaration of the template. If two or more formal subprograms of the template have the same defining name, then named associations are not allowed for the corresponding actuals.

Section 13: Representation Issues

13.1 Representation Items

Insert after paragraph 9.1: [AI05-0106-1]

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

the new paragraph:

Unless otherwise specified, an operational or representational item shall not specify an aspect of a generic formal parameter.

13.3 Representation Attributes

Insert after paragraph 11: [AI05-0095-1]

Denotes the address of the first of the storage elements allocated to X. For a program unit or label, this value refers to the machine code associated with the corresponding body or statement. The value of this attribute is of type System.Address.

the new paragraph:

The prefix of X'Address shall not statically denote a subprogram that has convention Intrinsic. X'Address raises Program_Error if X denotes a subprogram that has convention Intrinsic.

13.9 Unchecked Type Conversions

Replace paragraph 7: [AI05-0078-1]

- S'Alignment = Target'Alignment.

by:

- S'Alignment is a multiple of Target'Alignment or Target'Alignment is zero.

13.9.1 Data Validity

Replace paragraph 13: [AI05-0054-2]

The dereference of an access value is erroneous if it does not designate an object of an appropriate type or a subprogram with an appropriate profile, if it designates a nonexistent object, or if it is an access-to-variable value that designates a constant object. Such an access value can exist, for example, because of Unchecked_Deallocation, Unchecked_Access, or Unchecked_Conversion.

by:

The dereference of an access value is erroneous if it does not designate an object of an appropriate type or a subprogram with an appropriate profile, if it designates a nonexistent object, or if it is an access-to-variable value that designates a constant object and it did not originate from an attribute_reference applied to an aliased variable view of a controlled or immutably limited object. An access value whose dereference is erroneous can exist, for example, because of Unchecked_Deallocation, Unchecked_Access, or Unchecked_Conversion.

13.11.2 Unchecked Storage Deallocation

Replace paragraph 16: [AI05-0033-1]

Evaluating a name that denotes a nonexistent object is erroneous. The execution of a call to an instance of Unchecked_Deallocation is erroneous if the object was created other than by an allocator for an access type whose pool is Name'Storage_Pool.

by:

Evaluating a name that denotes a nonexistent object or a protected subprogram whose associated object is nonexistent is erroneous. The execution of a call to an instance of Unchecked_Deallocation is erroneous if the object was created other than by an allocator for an access type whose pool is Name'Storage_Pool.

13.12 Pragma Restrictions

Replace paragraph 8: [AI05-0013-1]

A pragma Restrictions is a configuration pragma; unless otherwise specified for a particular restriction, a partition shall obey the restriction if a pragma Restrictions applies to any compilation unit included in the partition.

by:

A pragma Restrictions is a configuration pragma. If a pragma Restrictions applies to any compilation unit included in the partition, this may impose either (or both) of two kinds of requirements, as specified for the particular restriction:

- A restriction may impose requirements on some or all of the units comprising the partition. Unless otherwise specified for a particular restriction, such a requirement applies to all of the units comprising the partition and is enforced via a post-compilation check.
- A restriction may impose requirements on the runtime behavior of the program, as indicated by the specification of runtime behavior associated with a violation of the requirement.

13.13.2 Stream-Oriented Attributes

Insert after paragraph 9: [AI05-0023-1]

For elementary types, Read reads (and Write writes) the number of stream elements implied by the Stream_Size for the type *T*; the representation of those stream elements is implementation defined. For composite types, the Write or Read attribute for each component is called in canonical order, which is last dimension varying fastest for an array, and positional aggregate order for a record. Bounds are not included in the stream if *T* is an array type. If *T* is a discriminated type, discriminants are included only if they have defaults. If *T* is a tagged type, the tag is not included. For type extensions, the Write or Read attribute for the parent type is called, followed by the Write or Read attribute of each component of the extension part, in canonical order. For a limited type extension, if the attribute of the parent type or any progenitor type of *T* is available anywhere within the immediate scope of *T*, and the attribute of the parent type or the type of any of the extension components is not available at the freezing point of *T*, then the attribute of *T* shall be directly specified.

the new paragraph:

If *T* is a discriminated type and its discriminants have defaults then S'Read first reads the discriminants from the stream without modifying *Item*. S'Read then creates an object of type *T* constrained by these discriminants. The value of this object is then converted to the subtype of *Item* and is assigned to *Item*. Finally, the Read attribute for each non-discriminant component of *Item* is called in canonical order as described above. Normal default initialization and finalization take place for the created object.

Replace paragraph 27: [AI05-0023-1]

- S'Output then calls S'Write to write the value of *Item* to the stream. S'Input then creates an object (with the bounds or discriminants, if any, taken from the stream), passes it to S'Read, and returns the value of the object. Normal default initialization and finalization take place for this object (see 3.3.1, 7.6, and 7.6.1).

by:

- S'Output then calls S'Write to write the value of *Item* to the stream. S'Input then creates an object of type *T*, with the bounds or (when without defaults) the discriminants, if any, taken from the stream, passes it to S'Read, and returns the value of the object. If *T* has discriminants, then this object is unconstrained if and only the discriminants have defaults. Normal default initialization and finalization take place for this object (see 3.3.1, 7.6, and 7.6.1).

Replace paragraph 34: [AI05-0109-1]

First reads the external tag from *Stream* and determines the corresponding internal tag (by calling Tags.Descendant_Tag(String'Input(*Stream*), S'Tag) which might raise Tag_Error — see 3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that result. If the specific type identified by the internal tag is not covered by *T*'Class or is abstract, Constraint_Error is raised.

by:

First reads the external tag from *Stream* and determines the corresponding internal tag (by calling Tags.Descendant_Tag(String'Input(*Stream*), S'Tag) which might raise Tag_Error — see 3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that result. If the specific type identified by the internal tag is abstract, Constraint_Error is raised.

Replace paragraph 38: [AI05-0039-1]

The stream-oriented attributes may be specified for any type via an attribute_definition_clause. The subprogram name given in such a clause shall not denote 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. 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.

Replace paragraph 51: [AI05-0007-1]

For an attribute_definition_clause specifying one of these attributes, the subtype of the Item parameter shall be the base subtype if scalar, and the first subtype otherwise. The same rule applies to the result of the Input function.

by:

For an attribute_definition_clause specifying one of these attributes, the subtype of the *Item* parameter shall be the first subtype or the base subtype if scalar, and the first subtype if not scalar. The same rule applies to the result of the Input function.

Replace paragraph 52: [AI05-0065-1]

A type is said to *support external streaming* if Read and Write attributes are provided for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling the representation. A limited type supports external streaming only if it has available Read and Write attributes. A type with a part that is of an access type supports external streaming only if that access type or the type of some part that includes the access type component, has Read and Write attributes that have been specified via an attribute_definition_clause, and that attribute_definition_clause is visible. An anonymous access type does not support external streaming. All other types support external streaming.

by:

A type is said to *support external streaming* if Read and Write attributes are provided for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling the representation. A limited type supports external streaming only if it has available Read and Write attributes. A type with a part that is of a non-remote access type supports external streaming only if that access type or the type of some part that includes the access type component, has Read and Write attributes that have been specified via an attribute_definition_clause, and that attribute_definition_clause is visible. An anonymous access type does not support external streaming. All other types (including remote access types, see E.2.2) support external streaming.

Insert after paragraph 56: [AI05-0023-1]

The number of calls performed by the predefined implementation of the stream-oriented attributes on the Read and Write operations of the stream type is unspecified. An implementation may take advantage of this permission to perform internal buffering. However, all the calls on the Read and Write operations of the stream type needed to implement an explicit invocation of a stream-oriented attribute must take place before this invocation returns. An explicit invocation is one appearing explicitly in the program text, possibly through a generic instantiation (see 12.3).

the new paragraphs:

If *T* is a discriminated type and its discriminants have defaults then in two cases an execution of the default implementation of S'Read is not required to create an anonymous object of type *T*: If the discriminants values that are read in are equal to the corresponding discriminant values of *Item*, then no object of type *T* need be created and *Item* may be used instead. If they are not equal and *Item* is a constrained variable, then Constraint_Error may be raised at that point, before any further values are read from the stream and before the object of type *T* is created.

A default implementation of S'Input that calls the default implementation of S'Read may create a constrained anonymous object with discriminants that match those in the stream.

13.14 Freezing Rules

Insert after paragraph 2: [AI05-0019-1]

The *freezing* of an entity occurs at one or more places (*freezing points*) in the program text where the representation for the entity has to be fully determined. Each entity is frozen from its first freezing point to the end of the program text (given the ordering of compilation units defined in 10.1.4).

the new paragraph:

This clause also defines a place in the program text where the profile of each declared callable entity becomes *frozen*. A use of a callable entity causes freezing of its profile in some contexts, as described below. At the place where the profile of a callable entity becomes frozen, the entity itself becomes frozen.

Replace paragraph 3: [AI05-0017-1; AI05-0019-1]

The end of a declarative_part, protected_body, or a declaration of a library package or generic library package, causes *freezing* of each entity declared within it, except for incomplete types. A noninstance body other than a renames-as-body causes

freezing of each entity declared before it within the same declarative_part.

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 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.

Replace paragraph 8.1: [AI05-0019-1]

An implicit call freezes the same entities that would be frozen by an explicit call. This is true even if the implicit call is removed via implementation permissions.

by:

An implicit call freezes the same entities and profiles that would be frozen by an explicit call. This is true even if the implicit call is removed via implementation permissions.

Insert after paragraph 10: [AI05-0019-1]

- At the place where an expression causes freezing, the type of the expression is frozen, unless the expression is an enumeration literal used as a discrete_choice of the array_aggregate of an enumeration_representation_clause.

the new paragraphs:

- 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.
- At the place where a generic_instantiation causes freezing of a callable entity, the profile of that entity is frozen.

Replace paragraph 14: [AI05-0019-1]

At the place where a callable entity is frozen, each subtype of its profile is frozen. If the callable entity is a member of an entry family, the index subtype of the family is frozen. At the place where a function call causes freezing, if a parameter of the call is defaulted, the default_expression for that parameter causes freezing.

by:

At the place where a profile is frozen, each subtype of the profile is frozen. If the corresponding callable entity is a member of an entry family, the index subtype of the family is frozen.

Replace paragraph 15.1: [AI05-0019-1]

At the place where a specific tagged type is frozen, the primitive subprograms of the type are frozen.

by:

At the place where a specific tagged type is frozen, the primitive subprograms of the type are frozen. At the place where a type is frozen, any subprogram named in an attribute_definition_clause for the type is frozen.

Annex A: Predefined Language Environment

Insert after paragraph 3: [AI05-0048-1]

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.

the new paragraph:

If a descendent of a language-defined tagged type is declared, the implementation shall ensure that each inherited language-defined subprogram behaves as described in this International Standard. In particular, overriding a language-defined subprogram shall not alter the effect of any inherited language-defined subprogram.

Insert after paragraph 4: [AI05-0060-1]

The implementation may restrict the replacement of language-defined compilation units. The implementation may restrict children of language-defined library units (other than Standard).

the new paragraph:

An implementation may omit pragma Remote_Types (see E.2.2) in language-defined library units if the implementation does not conform to Annex E.

A.4.3 Fixed-Length String Handling

Replace paragraph 56.2: [AI05-0056-1]

Each Index function searches, starting from From, for a slice of Source, with length Pattern'Length, that matches Pattern with respect to Mapping; the parameter Going indicates the direction of the lookup. If From is not in Source'Range, then Index_Error is propagated. If Going = Forward, then Index returns the smallest index I which is greater than or equal to From such that the slice of Source starting at I matches Pattern. If Going = Backward, then Index returns the largest index I such that the slice of Source starting at I matches Pattern and has an upper bound less than or equal to From. If there is no such slice, then 0 is returned. If Pattern is the null string, then Pattern_Error is propagated.

by:

Each Index function searches, starting from From, for a slice of Source, with length Pattern'Length, that matches Pattern with respect to Mapping; the parameter Going indicates the direction of the lookup. If Source is the null string, Index returns 0; otherwise, if From is not in Source'Range, then Index_Error is propagated. If Going = Forward, then Index returns the smallest index I which is greater than or equal to From such that the slice of Source starting at I matches Pattern. If Going = Backward, then Index returns the largest index I such that the slice of Source starting at I matches Pattern and has an upper bound less than or equal to From. If there is no such slice, then 0 is returned. If Pattern is the null string, then Pattern_Error is propagated.

A.10.5 Operations on Columns, Lines, and Pages

Replace paragraph 37: [AI05-0038-1]

- If the value specified by To is greater than the current line number, has the effect of repeatedly calling New_Line (with a spacing of one), until the current line number equals the specified value. If the value specified by To is equal to the current line number, there is no effect. If the value specified by To is less than the current line number, has the effect of calling New_Page followed by a call of New_Line with a spacing equal to (To – 1).

by:

- If the value specified by To is greater than the current line number, has the effect of repeatedly calling New_Line (with a spacing of one), until the current line number equals the specified value. If the value specified by To is equal to the current line number, there is no effect. If the value specified by To is less than the current line number, has the effect of calling New_Page followed, if To is greater than 1, by a call of New_Line with a spacing equal to (To – 1).

A.10.7 Input-Output of Characters and Strings

Replace paragraph 8: [AI05-0038-1]

Mode_Error is propagated if the mode of the file is not In_File. Sets End_Of_Line to True if at end of line, including if at end of page or at end of file; in each of these cases the value of Item is not specified. Otherwise End_Of_Line is set to False and Item is set to the the next character (without consuming it) from the file.

by:

Status_Error is propagated if the file is not open. Mode_Error is propagated if the mode of the file is not In_File. Sets End_Of_Line to True if at end of line, including if at end of page or at end of file; in each of these cases the value of Item is not specified. Otherwise End_Of_Line is set to False and Item is set to the the next character (without consuming it) from the file.

Replace paragraph 10: [AI05-0038-1]

Reads the next character, either control or graphic, from the specified File or the default input file. Mode_Error is propagated if the mode of the file is not In_File. End_Error is propagated if at the end of the file. The current column, line and page numbers for the file are not affected.

by:

Reads the next character, either control or graphic, from the specified File or the default input file. Status_Error is propagated if the file is not open. Mode_Error is propagated if the mode of the file is not In_File. End_Error is propagated if at the end of the file. The current column, line and page numbers for the file are not affected.

Replace paragraph 12: [AI05-0038-1]

If a character, either control or graphic, is available from the specified File or the default input file, then the character is read; Available is True and Item contains the value of this character. If a character is not available, then Available is False and the value of Item is not specified. Mode_Error is propagated if the mode of the file is not In_File. End_Error is propagated if at the end of the file. The current column, line and page numbers for the file are not affected.

by:

If a character, either control or graphic, is available from the specified File or the default input file, then the character is read; Available is True and Item contains the value of this character. If a character is not available, then Available is False and the value of Item is not specified. Status_Error is propagated if the file is not open. Mode_Error is propagated if the mode of the file is not In_File. End_Error is propagated if at the end of the file. The current column, line and page numbers for the file are not affected.

A.10.8 Input-Output for Integer Types

Replace paragraph 10: [AI05-0038-1]

The exception Data_Error is propagated if the sequence of characters read does not form a legal integer literal or if the value obtained is not of the subtype Num (for Integer_IO) or is not in the base range of Num (for Modular_IO).

by:

The exception Data_Error is propagated if the sequence of characters read does not form a legal integer literal or if the value obtained is not of the subtype Num.

Delete paragraph 24: [AI05-0038-1]

30 For Modular_IO, execution of Get propagates Data_Error if the sequence of characters read forms an integer literal outside the range 0..Num'Last.

A.11 Wide Text Input-Output and Wide Wide Text Input-Output

Replace paragraph 4: [AI05-0004-1]

The specification of package Wide_Text_IO.Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Wide_Bounded_String, and any occurrence of package Bounded is replaced by Wide_Bounded. The specification of package Wide_Wide_Text_IO.Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Wide_Wide_Bounded_String, and any occurrence of package Bounded is replaced by Wide_Wide_Bounded.

by:

The specification of package Wide_Text_IO.Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Wide_Bounded_String, and any occurrence of package Bounded is replaced by Wide_Bounded. The specification of package Wide_Wide_Text_IO.Wide_Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Wide_Wide_Bounded_String, and any occurrence of package Bounded is replaced by Wide_Wide_Bounded.

A.18 Containers

Insert after paragraph 4: [AI05-0044-1]

If the advice suggests that the complexity should be less than $O(f(N))$, then for any arbitrarily small positive real D, there should exist a positive integer M such that for all $N > M$, $t(N)/f(N) < D$.

the new paragraph:

When a formal operator "<" is used to provide an ordering for a container, it is generally required to define a strict weak ordering. An operator "<" 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 $(z < y)$.

A.18.2 The Package Containers.Vectors

Replace paragraph 6: [AI05-0084-1]

```
generic
   type Index_Type is range <>;
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Vectors is
   pragma Preelaborate(Vectors);
```

by:

```
generic
   type Index_Type is range <>;
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Vectors is
   pragma Preelaborate(Vectors);
   pragma Remote_Types(Vectors);
```

Replace paragraph 139: [AI05-0021-1]

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

by:

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of the vector that contains the element designated by Cursor. Any exception raised by Process.**all** is propagated.

Replace paragraph 231: [AI05-0044-1]

The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. How many times the subprograms of Generic_Sorting call "<" is unspecified.

by:

The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. How many times the subprograms of Generic_Sorting call "<" is unspecified.

Replace paragraph 237: [AI05-0021-1]

Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated.

by:

If Source is empty, then Merge does nothing. If Source and Target are the same non-empty container object, then Program_Error is propagated. Otherwise, Merge removes elements from Source and inserts them into Target;

afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated.

Insert after paragraph 239: [AI05-0022-1; AI05-0027-1]

Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined.

the new paragraph:

It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of this package, to tamper with elements of any Vector parameter to the operation. Either Program_Error is raised, or the operation works as defined on the value of the Vector either prior to, or subsequent to, some or all of the modifications to the Vector.

It is a bounded error to call any subprogram declared in the visible part of Containers.Vectors when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

A.18.3 The Package Containers.Doubly_Linked_Lists

Replace paragraph 5: [AI05-0084-1]

```
generic
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists is
   pragma Preelaborate(Doubly_Linked_Lists);
```

by:

```
generic
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists is
   pragma Preelaborate(Doubly_Linked_Lists);
   pragma Remote_Types(Doubly_Linked_Lists);
```

Replace paragraph 83: [AI05-0021-1]

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

by:

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of the list that contains the element designated by Cursor. Any exception raised by Process.**all** is propagated.

Replace paragraph 102: [AI05-0021-1]

Equivalent to Delete (Container, First (Container), Count).

by:

If Container is empty, Constraint_Error is propagated. Otherwise, Delete_First removes (from Container) Count elements starting at the first element in Container (or all of the elements if there are fewer than Count elements in Container).

Replace paragraph 145: [AI05-0044-1]

The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. How many times the subprograms of Generic_Sorting call "<" is unspecified.

by:

The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are

unspecified. How many times the subprograms of Generic_Sorting call "<" is unspecified.

Replace paragraph 151: [AI05-0021-1]

Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated.

by:

If Source is empty, then Merge does nothing. If Source and Target are the same non-empty container object, then Program_Error is propagated. Otherwise, Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated.

Insert after paragraph 152: [AI05-0022-1; AI05-0027-1]

Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined.

the new paragraphs:

It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of this package, to tamper with elements of any List parameter to the operation. Either Program_Error is raised, or the operation works as defined on the value of the List either prior to, or subsequent to, some or all of the modifications to the List.

It is a bounded error to call any subprogram declared in the visible part of Containers.Doubly_Linked_Lists when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

A.18.4 Maps

Replace paragraph 38: [AI05-0021-1]

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

by:

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Program_Error is propagated if Process.**all** tampers with the elements of the map that contains the element designated by Cursor. Any exception raised by Process.**all** is propagated.

Insert after paragraph 75: [AI05-0022-1; AI05-0027-1]

Iterate calls Process.**all** with a cursor that designates each node in Container, starting with the first node and moving the cursor according to the successor relation. Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process.**all** is propagated.

the new paragraphs:

Bounded Error

It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of a map package, to tamper with elements of any map parameter to the operation. Either Program_Error is raised, or the operation works as defined on the value of the map either prior to, or subsequent to, some or all of the modifications to the map.

It is a bounded error to call any subprogram declared in the visible part of a map package when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

A.18.5 The Package Containers.Hashed_Maps

Replace paragraph 2: [AI05-0084-1]

```
generic
   type Key_Type is private;
   type Element_Type is private;
   with function Hash (Key : Key_Type) return Hash_Type;
   with function Equivalent_Keys (Left, Right : Key_Type)
      return Boolean;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Hashed_Maps is
   pragma Preelaborate(Hashed_Maps);
```

by:

```
generic
   type Key_Type is private;
   type Element_Type is private;
   with function Hash (Key : Key_Type) return Hash_Type;
   with function Equivalent_Keys (Left, Right : Key_Type)
      return Boolean;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Hashed_Maps is
   pragma Preelaborate(Hashed_Maps);
   pragma Remote_Types(Hashed_Maps);
```

A.18.6 The Package Containers.Ordered_Maps

Replace paragraph 2: [AI05-0084-1]

```
generic
   type Key_Type is private;
   type Element_Type is private;
   with function "<" (Left, Right : Key_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Maps is
   pragma Preelaborate(Ordered_Maps);
```

by:

```
generic
   type Key_Type is private;
   type Element_Type is private;
   with function "<" (Left, Right : Key_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Maps is
   pragma Preelaborate(Ordered_Maps);
   pragma Remote_Types(Ordered_Maps);
```

Replace paragraph 56: [AI05-0044-1]

The actual function for the generic formal function "<" on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive. If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.

by:

The actual function for the generic formal function "<" on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict weak ordering relationship (see A.18). If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.

A.18.7 Sets

Replace paragraph 36: [AI05-0021-1]

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of Container. Any exception raised by Process.**all** is propagated.

by:

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.**all** with the element designated by Position as the argument. Program_Error is propagated if Process.**all** tampers with the elements of the set that contains the element designated by Cursor. Any exception raised by Process.**all** is propagated.

Replace paragraph 58: [AI05-0004-1]

Union deletes from Target the elements of Target that are not equivalent some element of Source.

by:

Intersection deletes from Target the elements of Target that are not equivalent some element of Source.

Delete paragraph 79: [AI05-0004-1]

Equivalent to Find (Container, Item) /= No_Element.

Insert after paragraph 82: [AI05-0004-1]

```
function Contains (Container : Set;
                   Item      : Element_Type) return Boolean;
```

the new paragraph:

Equivalent to Find (Container, Item) /= No_Element.

Insert after paragraph 96: [AI05-0022-1; AI05-0027-1]

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

the new paragraphs:

Bounded Error

It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of a set package, to tamper with elements of any set parameter to the operation. Either Program_Error is raised, or the operation works as defined on the value of the set either prior to, or subsequent to, some or all of the modifications to the set.

It is a bounded error to call any subprogram declared in the visible part of a set package when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

A.18.8 The Package Containers.Hashed_Sets

Replace paragraph 2: [AI05-0084-1]

```
generic
   type Element_Type is private;
   with function Hash (Element : Element_Type) return Hash_Type;
   with function Equivalent_Elements (Left, Right : Element_Type)
                 return Boolean;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Hashed_Sets is
   pragma Preelaborate(Hashed_Sets);
```

by:

```
generic
   type Element_Type is private;
   with function Hash (Element : Element_Type) return Hash_Type;
   with function Equivalent_Elements (Left, Right : Element_Type)
                 return Boolean;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Hashed_Sets is
   pragma Preelaborate(Hashed_Sets);
   pragma Remote_Types(Hashed_Sets);
```

Insert after paragraph 66: [AI05-0044-1]

The actual function for the generic formal function Equivalent_Elements is expected to return the same value each time it is called with a particular pair of Element values. It should define an equivalence relationship, that is, be reflexive, symmetric, and transitive. If the actual for Equivalent_Elements behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Equivalent_Elements, and how many times they call it, is unspecified.

the new paragraph:

If the actual function for the generic formal function "=" returns True for any pair of non-equivalent elements, then the behavior of the container function "=" is unspecified.

A.18.9 The Package Containers.Ordered_Sets

Replace paragraph 2: [AI05-0084-1]

```
generic
   type Element_Type is private;
   with function "<" (Left, Right : Element_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Sets is
   pragma Preelaborate(Ordered_Sets);
```

by:

```
generic
   type Element_Type is private;
   with function "<" (Left, Right : Element_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Sets is
   pragma Preelaborate(Ordered_Sets);
   pragma Remote_Types(Ordered_Sets);
```

Replace paragraph 79: [AI05-0044-1]

The actual function for the generic formal function "<" on Element_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive. If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.

by:

The actual function for the generic formal function "<" on Element_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict weak ordering relationship (see A.18). If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.

If the actual function for the generic formal function "=" returns True for any pair of non-equivalent elements, then the behavior of the container function "=" is unspecified.

A.18.16 Array Sorting

Replace paragraph 5: [AI05-0044-1]

The actual function for the generic formal function "<" of Generic_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Array_Sort is unspecified. How many times Generic_Array_Sort calls "<" is unspecified.

by:

The actual function for the generic formal function "<" of Generic_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Array_Sort is unspecified. How many times Generic_Array_Sort calls "<" is unspecified.

Replace paragraph 9: [AI05-0044-1]

The actual function for the generic formal function "<" of Generic_Constrained_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Constrained_Array_Sort is unspecified. How many times Generic_Constrained_Array_Sort calls "<" is unspecified.

by:

The actual function for the generic formal function "<" of Generic_Constrained_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Constrained_Array_Sort is unspecified. How many times Generic_Constrained_Array_Sort calls "<" is unspecified.

A.18.17 The Package Containers.Indefinite_Holders

Insert new clause: [AI05-0069-1; AI05-0084-1]

The language-defined generic package Containers.Indefinite_Holders provides private type Holder and a set of operations for that type. A holder container holds a single element of an indefinite type.

A holder containers allows the declaration of an object that can be used like an uninitialized variable or component of an indefinite type.

A holder container may be *empty*. An empty holder does not contain an element.

Static Semantics

The generic library package Containers.Indefinite_Holders has the following declaration:

```
generic
    type Element_Type (<>) is private;
    with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Indefinite_Holders is
    pragma Preelaborate (Indefinite_Holders);
    pragma Remote_Types (Indefinite_Holders);

    type Holder is tagged private;
    pragma Preelaborable_Initialization (Holder);

    Empty_Holder : constant Holder;

    function "=" (Left, Right : Holder) return Boolean;

    function To_Holder (New_Item : Element_Type) return Holder;

    function Is_Empty (Container : Holder) return Boolean;

    procedure Clear (Container : in out Holder);

    function Element (Container : Holder) return Element_Type;

    procedure Replace_Element (Container : in out Holder;
                               New_Item  : in     Element_Type);

    procedure Query_Element (Container : in Holder;
                             Process   : not null access procedure (Element : in
Element_Type));

    procedure Update_Element (Container : in Holder;
                              Process   : not null access procedure (Element : in out
Element_Type));

    procedure Move (Target : in out Holder; Source : in out Holder);

private

   ... -- not specified by the language

end Ada.Containers.Indefinite_Holders;
```

The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on holder values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on holder values are unspecified.

The type Holder is used to represent holder containers. The type Holder needs finalization (see 7.6).

Empty_Holder represents an empty holder object. If an object of type Holder is not otherwise initialized, it is initialized to the same value as Empty_Holder.

Some operations of this generic package have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for "tampering with elements" of a container because they depend on elements of the container not being replaced.

A subprogram is said to *tamper with elements* of a holder object *H* if:

- It clears the element contained by *H*, that is, it calls the Clear procedure with *H* as a parameter;

- It replaces the element contained by *H*, that is, it calls the Replace_Element procedure with *H* as a parameter;
- It calls the Move procedure with *H* as a parameter;
- It finalizes *H*.

```
function "=" (Left, Right : Holder) return Boolean;
```

If Left and Right denote the same holder object, then the function returns True. Otherwise, it compares the element contained in Left to the element contained in Right using the generic formal equality operator, returning the result of that operation. Any exception raised during the evaluation of element equality is propagated.

```
function To_Holder (New_Item : Element_Type) return Holder;
```

Returns a non-empty holder containing an element initialized to New_Item.

```
function Is_Empty (Container : Holder) return Boolean;
```

Returns True if the holder is empty, and False if it contains an element.

```
procedure Clear (Container : in out Holder);
```

Removes the element from Container. Container is empty after a successful Clear operation.

```
function Element (Container : Holder) return Element_Type;
```

If Container is empty, Constraint_Error is propagated. Otherwise, returns the element stored in Container.

```
procedure Replace_Element (Container : in out Holder;
                           New_Item  : in     Element_Type);
```

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.

```
procedure Query_Element (Container : in Holder;
                         Process   : not null access procedure (Element : in Element_Type));
```

If Container is empty, Constraint_Error is propagated. Otherwise, Query_Element calls Process.all with the contained element as the argument. Program_Error is raised if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated.

```
procedure Update_Element (Container : in Holder;
                          Process   : not null access procedure (Element : in out
Element_Type));
```

If Container is empty, Constraint_Error is propagated. Otherwise, Query_Element calls Process.all with the contained element as the argument. Program_Error is raised if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated.

```
procedure Move (Target : in out Holder; Source : in out Holder);
```

If Target denotes the same object as Source, then Move has no effect. Otherwise, the element contained by Source (if any) is removed from Source and inserted into Target, replacing any preexisting content. Source is empty after a successful call to Move.

Bounded Errors

It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of this package, to tamper with elements of any Holder parameter to the operation. Either Program_Error is raised, or the operation works as defined on the value of the Holder either prior to, or subsequent to, some or all of the modifications to the Holder.

It is a bounded error to call any subprogram declared in the visible part of Containers.Indefinite_Holders when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

Implementation Requirements

No storage associated with a holder object shall be lost upon assignment or scope exit.

The execution of an assignment_statement for a holder shall have the effect of copying the element (if any) from the source holder object to the target holder object.

Implementation Advice

Move should not copy elements, and should minimize copying of internal data structures.

If an exception is propagated from a holder operation, no storage should be lost, nor should the element be removed from a holder unless specified by the operation.

Annex B: Interface to Other Languages

B.1 Interfacing Pragmas

Replace paragraph 17: [AI05-0002-1]

- T is an access-to-object type, and its designated type is *L*-compatible,

by:

- T is an access-to-object type, its designated type is *L*-compatible, and its designated subtype is not an unconstrained array subtype,

B.3 Interfacing with C and C++

Insert after paragraph 62: [AI05-0002-1]

An implementation may provide additional declarations in the C interface packages.

the new paragraphs:

An implementation need not support an interfacing pragma specifying convention C or C_Pass_By_Copy in the following cases:

- an Export or Convention pragma applied to a subprogram which has a parameter of an unconstrained array subtype;
- an interfacing pragma applied to a function with an unconstrained array result subtype;
- an interfacing pragma applied to an object whose nominal subtype is an unconstrained array subtype.

Replace paragraph 71.1: [AI05-0002-1]

An Ada parameter of a private type is passed as specified for the full view of the type.

by:

- An Ada parameter of a private type is passed as specified for the full view of the type.
- The rules of correspondence given above for parameters of mode **in** also apply to the return object of a function.

B.3.3 Pragma Unchecked_Union

Replace paragraph 9: [AI05-0026-1]

Any name that denotes a discriminant of an object of an unchecked union type shall occur within the declarative region of the type.

by:

Any name that denotes a discriminant of an object of an unchecked union type shall occur within the declarative region of the type, and shall not occur within a record_representation_clause.

Replace paragraph 10: [AI05-0026-1]

A component declared in a variant_part of an unchecked union type shall not have a controlled, protected, or task part.

by:

The type of a component declared in a variant_part of an unchecked union type shall not need finalization. In addition to the places where Legality Rules normally apply (see 12.3), this rule also applies in the private part of an instance of a generic unit. For an unchecked union type declared within the body of a generic unit, or within the body of any of its descendant library units, no part of the type of a component declared in a variant_part of the unchecked union type shall be of a formal private type or formal private extension declared within the formal part of the generic unit.

Annex C: Systems Programming

C.3.1 Protected Procedure Handlers

Replace paragraph 7: [AI05-0033-1]

The Attach_Handler pragma is only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library-level declaration.

by:

The Attach_Handler and Interrupt_Handler pragmas are only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library-level declaration, and shall not be declared within a generic body. 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.

Delete paragraph 8: [AI05-0033-1]

The Interrupt_Handler pragma is only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library-level declaration.

C.7.1 The Package Task_Identification

Replace paragraph 17: [AI05-0004-1]

It is a bounded error to call the Current_Task function from an entry body, or an interrupt handler, or finalization of a task attribute. Program_Error is raised, or an implementation-defined value of the type Task_Id is returned.

by:

It is a bounded error to call the Current_Task function from an @fa{entry_body}, or an interrupt handler, or finalization of a task attribute. Program_Error is raised, or an implementation-defined value of the type Task_Id is returned.

Annex D: Real-Time Systems

D.2.6 Earliest Deadline First Dispatching

Replace paragraph 26: [AI05-0055-1]

- the highest priority *P*, if any, less than the base priority of *T* such that one or more tasks are executing within a protected object with ceiling priority *P* and task *T* has an earlier deadline than all such tasks.

by:

- the highest priority *P*, if any, less than the base priority of *T* such that one or more tasks are executing within a protected object with ceiling priority *P* and task *T* has an earlier deadline than all such tasks; and furthermore *T* has an earlier deadline than all other tasks on ready queues with priorities in the given EDF_Across_Priorities range that are strictly less than *P*.

D.7 Tasking Restrictions

Replace paragraph 3: [AI05-0013-1]

All (nonenvironment) tasks depend directly on the environment task of the partition.

by:

No task depends on a task other than the environment task of the partition.

Replace paragraph 4: [AI05-0013-1]

Objects of a type that needs finalization (see 7.6) and access types that designate a type that needs finalization, shall be declared only at library level.

by:

Objects of a type that needs finalization (see 7.6) are declared only at library level. If an access type does not have library-level accessibility, then there are no allocators of the type where the type determined by the subtype_mark of the subtype_indication or qualified_expression needs finalization.

Replace paragraph 10: [AI05-0013-1]

There is no call to any of the operations defined in package Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference).

by:

There is no use of any of the operations defined in package Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference).

Replace paragraph 10.1: [AI05-0013-1]

Protected objects shall be declared only at library level.

by:

Protected objects are declared only at library level.

Replace paragraph 10.2: [AI05-0013-1]

Timing_Events shall be declared only at library level.

by:

Timing_Events are declared only at library level.

Replace paragraph 10.8: [AI05-0013-1]

The Boolean expression in an entry barrier shall be either a static Boolean expression or a Boolean component of the enclosing protected object.

by:

The Boolean expression in each entry barrier is either a static expression or a name that statically denotes a component of the enclosing protected object.

D.9 Delay Accuracy

Replace paragraph 6: [AI05-0004-1]

When a delay_statement appears in a delay_alternative of a timed_entry_call the selection of the entry call is attempted, regardless of the specified expiration time. When a delay_statement appears in a selective_accept_alternative, and a call is queued on one of the open entries, the selection of that entry call proceeds, regardless of the value of the delay expression.

by:

When a delay_statement appears in a delay_alternative of a timed_entry_call the selection of the entry call is attempted, regardless of the specified expiration time. When a delay_statement appears in a select_alternative, and a call is queued on one of the open entries, the selection of that entry call proceeds, regardless of the value of the delay expression.

D.15 Timing Events

Replace paragraph 15: [AI05-0094-1]

If a procedure Set_Handler is called with zero or negative In_Time or with At_Time indicating a time in the past then the handler is executed immediately by the task executing the call of Set_Handler. The timing event Event is cleared.

by:

If a procedure Set_Handler is called with zero or negative In_Time or with At_Time indicating a time in the past then the handler is executed as soon as possible after the completion of the call of Set_Handler.

Annex E: Distributed Systems

E.2.2 Remote Types Library Units

Replace paragraph 9: [AI05-0060-1]

An access type declared in the visible part of a remote types or remote call interface library unit is called a *remote access type*. Such a type shall be:

by:

A named access type declared in the visible part of a remote types or remote call interface library unit is called a *remote access type*. Such a type shall be:

Replace paragraph 9.2: [AI05-0060-1]

- a general access type that designates a class-wide limited private type or a class-wide private type extension all of whose ancestors are either private type extensions or limited private types.

by:

- a general access type that designates a class-wide limited private type, a class-wide limited interface type, or a class-wide private extension all of whose ancestors are either private extensions, limited interface types, or limited private types.

Replace paragraph 14: [AI05-0060-1; AI05-0101-1]

- The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; each non-controlling formal parameter shall support external streaming (see 13.13.2);

by:

- The primitive subprograms of the corresponding specific type shall only have access parameters if they are controlling formal parameters. The primitive functions of the corresponding specific type shall only have an access result if it is a controlling access result. Each non-controlling formal parameter and non-controlling result subtype shall support external streaming (see 13.13.2);
- The corresponding specific type shall not have a primitive procedure to which a pragma Implemented applies unless the implementation_kind of the pragma is By_Any;

Insert after paragraph 16: [AI05-0101-1]

- A value of a remote access-to-class-wide type shall be dereferenced (or implicitly converted to an anonymous access type) only as part of a dispatching call where the value designates a controlling operand of the call (see E.4, "Remote Subprogram Calls");

the new paragraph:

- A controlling access result value for a primitive function with any controlling operands of the corresponding specific type shall either be explicitly converted to a remote access-to-class-wide type or be part of a dispatching call where the value designates a controlling operand of the call;

Insert after paragraph 18: [AI05-0060-1]

A remote types library unit need not be pure, and the types it defines may include levels of indirection implemented by using access types. User-specified Read and Write attributes (see 13.13.2) provide for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling any levels of indirection.

the new paragraphs:

The value of a remote access-to-class-wide limited interface can designate an object of a nonlimited type derived from the interface.

A remote access type may designate a class-wide synchronized, protected, or task interface type.

E.4 Remote Subprogram Calls

Replace paragraph 7: [AI05-0101-1]

A remote call interface library unit (see E.2.3) defines the remote subprograms or remote access types used for remote subprogram calls.

Remote types library units (see E.2.2) and remote call interface library units (see E.2.3) define the remote subprograms or remote access types used for remote subprogram calls.

Annex F: Information Systems

No changes in this section.

Annex G: Numerics

G.3.1 Real Vectors and Matrices

Replace paragraph 78: [AI05-0047-1]

This procedure computes both the eigenvalues and eigenvectors of the symmetric matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are normalized and mutually orthogonal (they are orthonormal), including when there are repeated eigenvalues. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index ranges of the parameter Vectors are those of A. Argument_Error is raised if the matrix A is not symmetric.

by:

This procedure computes both the eigenvalues and eigenvectors of the symmetric matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are normalized and mutually orthogonal (they are orthonormal), including when there are repeated eigenvalues. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2), or if Values'Range is not equal to A'Range(1), or if the index ranges of the parameter Vectors are not equal to those of A. Argument_Error is raised if the matrix A is not symmetric. Constraint_Error is also raised in implementation-defined circumstances if the algorithm used does not converge quickly enough.

Insert after paragraph 90: [AI05-0047-1]

The test that a matrix is symmetric should be performed by using the equality operator to compare the relevant components.

the new paragraph:

An implementation should minimize the circumstances under which the algorithm used for Eigenvalues and Eigensystem fails to converge.

G.3.2 Complex Vectors and Matrices

Replace paragraph 16: [AI05-0047-1]

```
function "abs" (Right : Complex_Vector) return Complex;
```

by:

```
function "abs" (Right : Complex_Vector) return Real'Base;
```

Replace paragraph 75: [AI05-0047-1]

```
function "abs" (Right : Complex_Vector) return Complex;
```

by:

```
function "abs" (Right : Complex_Vector) return Real'Base;
```

Replace paragraph 146: [AI05-0047-1]

This procedure computes both the eigenvalues and eigenvectors of the Hermitian matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are mutually orthonormal, including when there are repeated eigenvalues. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index ranges of the parameter Vectors are those of A. Argument_Error is raised if the matrix A is not Hermitian.

by:

This procedure computes both the eigenvalues and eigenvectors of the Hermitian matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are mutually orthonormal, including when there are repeated eigenvalues. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2), or if Values'Range is not equal to A'Range(1), or if the index ranges of the parameter Vectors are not equal to those of A. Argument_Error is raised if the matrix A is not Hermitian. Constraint_Error is also raised in implementation-defined circumstances if the algorithm used does not converge quickly enough.

Insert after paragraph 160: [AI05-0047-1]

The test that a matrix is Hermitian should use the equality operator to compare the real components and negation followed by equality to compare the imaginary components (see G.2.1).

the new paragraph:

An implementation should minimize the circumstances under which the algorithm used for Eigenvalues and Eigensystem fails to converge.

Annex H: Safety and Security

No changes in this section.

Annex J: Obsolescent Features

J.1 Renamings of Ada 83 Library Units

Replace the title: [AI05-0004-1]

Renamings of Ada 83 Library Units

by:

Renamings of Library Units

Annex M: Implementation-Defined Characteristics

No changes in this section.

Annex N: Glossary

No changes in this section.

Annex Q: Language-Defined Entities

No changes in this section.