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.

© 2010, 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 Any proposal may be substantially changed or withdrawn before this document begins standardization, and other proposals may be added.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. It was amended byISO/IEC 8652:1995/AMD.1:2007, adding additional capabilities.

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

- The use and functionality of the predefined containers;
- The ability to write and enforce contracts for Ada entities;
- The capabilities of Ada on multi-core and multi-threaded architectures;
- The safety, use, and functionality of access types and dynamic storage management.

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

- Incomplete types are allowed to be used and completed in more contexts (see 3.10.1);
- Consistency is improved by equality composing for untagged record types (see 4.5.2), labels at the end of statements (see 5.1), and In Out parameters being provided for functions (see 6.1);
- Use of anonymous access types is simplified by additional conversions and memberships (see 4.5.2 and 8.6);
- If (see 4.5.7) and case (see 4.5.8) expressions;
- Generalizing extended return statements (see 6.5);
- Additional support for processing of hierarchical file names (see A.16 and A.16.1); 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 12: [AI05-0176-1]

- A vertical line separates alternative items unless it immediately occurs after an opening curly bracket, in which case it stands for itself:

by:

- A vertical line separates alternative items unless it immediately occurs after an opening curly bracket, a loop_parameter_specification, or an iterator_specification, in which case it stands for itself:

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".

1.2 Normative References

Insert after paragraph 1: [AI05-0127-2]

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

the new paragraph:

ISO/IEC 639-3:2007, *Terminology and other language and content resources — Codes for the representation of names of languages — Part 3: Alpha-3 code for comprehensive coverage of languages*

Insert after paragraph 4: [AI05-0127-2]

ISO/IEC 1989:2002, *Information technology — Programming languages — COBOL*

the new paragraph:

ISO/IEC 3166-1:2006, *Information and documentation — Codes for the representation of names of countries and their subdivisions — Part 1: Country Codes*

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 5.1: [AI05-0091-1]

- The characters in category other_format are eliminated.

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; AI05-0163-1]

- 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.
- In place of a statement in a sequence_of_statements.
- At any place where a compilation_unit is allowed.

Replace paragraph 16: [AI05-0163-1]

Normally, implementation-defined pragmas should have no semantic effect for error-free programs; that is, if the implementation-defined pragmas are removed from a working program, the program should still be legal, and should still have the same semantics.

by:

Normally, implementation-defined pragmas should have no semantic effect for error-free programs; that is, if the implementation-defined pragmas in a working program are replaced with unrecognized pragmas, the program should still be legal, and should still have the same semantics.

2.9 Reserved Words

Replace paragraph 2: [AI05-0091-1; AI05-0176-1]

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.

In the list in paragraph 2, add: [AI05-0091-1; AI05-0176-1]

some

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.

3.2.1 Type Declarations

Replace paragraph 3: [AI05-0183-1]

```
full_type_declaration ::=
    type defining_identifier [known_discriminant_part] is type_definition;
  | task_type_declaration
  | protected_type_declaration
```

by:

```
full_type_declaration ::=
    type defining_identifier [known_discriminant_part] is type_definition
      [aspect_specification];
  | task_type_declaration
  | protected_type_declaration
```

3.2.2 Subtype Declarations

Replace paragraph 2: [AI05-0183-1]

```
subtype_declaration ::=
  subtype defining_identifier is subtype_indication;
```

by:

```
subtype_declaration ::=
  subtype defining_identifier is subtype_indication
    [aspect_specification];
```

3.2.3 Classification of Operations

Insert after paragraph 6: [AI05-0128-1]

- For a specific type declared immediately within a package_specification, any subprograms (in addition to the enumeration literals) that are explicitly declared immediately within the same package_specification and that operate on the type;

the new paragraph:

- For a specific type with an explicitly declared primitive "=" operator whose result type is Boolean, the corresponding "/=" operator (see 6.6);

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).

3.3.1 Object Declarations

Replace paragraph 2: [AI05-0183-1]

object_declaration ::=
  defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression];
 | defining_identifier_list : [**aliased**] [**constant**] access_definition [:= expression];
 | defining_identifier_list : [**aliased**] [**constant**] array_type_definition [:= expression];
 | single_task_declaration
 | single_protected_declaration

by:

object_declaration ::=
  defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression]
    [aspect_specification];
 | defining_identifier_list : [**aliased**] [**constant**] access_definition [:= expression]
    [aspect_specification];
 | defining_identifier_list : [**aliased**] [**constant**] array_type_definition [:= expression]
    [aspect_specification];
 | single_task_declaration
 | single_protected_declaration

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.

Replace paragraph 18: [AI05-0164-1]

The profile of an inherited subprogram (including an inherited enumeration literal) is obtained from the profile of the corresponding (user-defined) primitive subprogram of the parent or progenitor type, after systematic replacement of each subtype of its profile (see 6.1) that is of the parent or progenitor type with a *corresponding subtype* of the derived type. For a given subtype of the parent or progenitor type, the corresponding subtype of the derived type is defined as follows:

by:

The profile of an inherited subprogram (including an inherited enumeration literal) is obtained from the profile of the corresponding (user-defined) primitive subprogram of the parent or progenitor type, after systematic replacement of each subtype of its profile (see 6.1), other than those found in the designated profile of an access_definition, that is of the parent or progenitor type with a *corresponding subtype* of the derived type. For a given subtype of the parent or progenitor type, the corresponding subtype of the derived type is defined as follows:

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.

3.5.2 Character Types

Replace paragraph 2: [AI05-0181-1]

The predefined type Character is a character type whose values correspond to the 256 code positions of Row 00 (also known as Latin-1) of the ISO/IEC 10646:2003 Basic Multilingual Plane (BMP). Each of the graphic characters of Row 00 of the BMP has a corresponding character_literal in Character. Each of the nongraphic positions of Row 00 (0000-001F and 007F-009F) has a corresponding language-defined name, which is not usable as an enumeration literal, but which is usable with the attributes Image, Wide_Image, Wide_Wide_Image, Value, Wide_Value, and Wide_Wide_Value; these names are given in the definition of type Character in A.1, "The Package Standard", but are set in *italics*.

by:

The predefined type Character is a character type whose values correspond to the 256 code positions of Row 00 (also known as Latin-1) of the ISO/IEC 10646:2003 Basic Multilingual Plane (BMP). Each of the graphic characters of Row 00 of the BMP has a corresponding character_literal in Character. Each of the nongraphic positions of Row 00 has a corresponding language-defined name, which is not usable as an enumeration literal, but which is usable with the attributes Image, Wide_Image, Wide_Wide_Image, Value, Wide_Value, and Wide_Wide_Value; these names are given in the definition of type Character in A.1, "The Package Standard", but are set in *italics*.

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.

Delete paragraph 16: [AI05-0102-1]

The type of the default_expression, if any, for an access discriminant shall be convertible to the anonymous access type of the discriminant (see 4.6).

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.

Delete paragraph 9: [AI05-0102-1]

The expression associated with an access discriminant shall be of a type convertible to the anonymous access type.

3.8 Record Types

Replace paragraph 6: [AI05-0183-1]

```
component_declaration ::=
  defining_identifier_list : component_definition [:= default_expression];
```

by:

```
component_declaration ::=
  defining_identifier_list : component_definition [:= default_expression]
    [aspect_specification];
```

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 Tagged Types and Type Extensions

Insert after paragraph 7.4: [AI05-0173-1]

```
function Interface_Ancestor_Tags (T : Tag) return Tag_Array;
```

the new paragraph:

```
function Is_Abstract (T : Tag) return Boolean;
```

Replace paragraph 12.1: [AI05-0113-1]

The function Descendant_Tag returns the (internal) tag for the type that corresponds to the given external tag and is both a descendant of the type identified by the Ancestor tag and has the same accessibility level as the identified ancestor. Tag_Error is raised if External is not the external tag for such a type. Tag_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created.

by:

The function Descendant_Tag returns the (internal) tag for the type that corresponds to the given external tag and is both a descendant of the type identified by the Ancestor tag and has the same accessibility level as the identified ancestor. Tag_Error is raised if External is not the external tag for such a type. Tag_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created, or if the given external tag identifies more than one type that has the appropriate Ancestor and accessibility level.

Insert after paragraph 12.4: [AI05-0173-1]

The function Interface_Ancestor_Tags returns an array containing the tag of each interface ancestor type of the type whose tag is T, other than T itself. The lower bound of the returned array is 1, and the order of the returned tags is unspecified. Each tag appears in the result exactly once. If the type whose tag is T has no interface ancestors, a null array is returned.

the new paragraph:

The function Is_Abstract returns True if the type whose tag is T is abstract, and False otherwise.

Replace paragraph 26.1: [AI05-0113-1]

Internal_Tag should return the tag of a type whose innermost master is the master of the point of the function call.

by:

Internal_Tag should return the tag of a type, if one exists, whose innermost master is a master of the point of the function call.

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 was inherited.

3.9.3 Abstract Types and Subprograms

Replace paragraph 1.1: [AI05-0183-1]

```
abstract_subprogram_declaration ::=
  [overriding_indicator]
  subprogram_specification is abstract;
```

by:

```
abstract_subprogram_declaration ::=
  [overriding_indicator]
  subprogram_specification is abstract
    [aspect_specification];
```

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; AI05-0203-1]

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. The type denoted by a return_subtype_indication (see 6.5) shall not 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; AI05-0142-4]

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, parameter_specification, 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 2.4: [AI05-0208-1]

- it occurs within the scope of a nonlimited_with_clause that mentions a library package in whose visible part the completion of *T* is declared.

by:

- it occurs within the scope of a nonlimited_with_clause that mentions a library package in whose visible part the completion of *T* is declared, or
- it occurs within the scope of the completion of *T* and *T* is an incomplete view declared by an incomplete_type_declaration.

Replace paragraph 2.5: [AI05-0162-1]

In these cases, the dereference has the full view of *T*.

by:

In these cases, the dereference has the view of *T* visible at the point of the dereference.

Replace paragraph 2.6: [AI05-0162-1; AI05-0208-1]

Similarly, if a subtype_mark denotes a subtype_declaration defining a subtype of an incomplete view *T*, the subtype_mark denotes an incomplete view except under the same two circumstances given above, in which case it denotes the full view of *T*.

by:

Similarly, if a subtype_mark denotes a subtype_declaration defining a subtype of an incomplete view *T*, the subtype_mark denotes an incomplete view except under the same three circumstances given above, in which case it denotes the view of *T* visible at the point of the subtype_mark.

Replace paragraph 3: [AI05-0162-1]

An incomplete_type_declaration requires a completion, which shall be a full_type_declaration. If the incomplete_type_declaration occurs immediately within either the visible part of a package_specification or a declarative_part, then the full_type_declaration shall occur later and immediately within this visible part or declarative_part. If the incomplete_type_declaration occurs immediately within the private part of a given package_specification, then the full_type_declaration shall occur later and immediately within either the private part itself, or the declarative_part of the corresponding package_body.

by:

An incomplete_type_declaration requires a completion, which shall be a type_declaration other than an incomplete_type_declaration. If the incomplete_type_declaration occurs immediately within either the visible part of a package_specification or a declarative_part, then the type_declaration shall occur later and immediately within this visible part or declarative_part. If the incomplete_type_declaration occurs immediately within the private part of a given package_specification, then the type_declaration shall occur later and immediately within either the private part itself, or the declarative_part of the corresponding package_body.

Replace paragraph 4: [AI05-0162-1]

If an incomplete_type_declaration includes the reserved word **tagged**, then a full_type_declaration that completes it shall declare a tagged type. If an incomplete_type_declaration has a known_discriminant_part, then a full_type_declaration that completes it shall have a fully conforming (explicit) known_discriminant_part (see 6.3.1). If an incomplete_type_declaration has no discriminant_part (or an unknown_discriminant_part), then a corresponding full_type_declaration is nevertheless allowed to have discriminants, either explicitly, or inherited via derivation.

by:

If an incomplete_type_declaration includes the reserved word **tagged**, then a type_declaration that completes it shall declare a tagged type. If an incomplete_type_declaration has a known_discriminant_part, then a type_declaration that completes it shall have a fully conforming (explicit) known_discriminant_part (see 6.3.1). If an incomplete_type_declaration has no discriminant_part (or an unknown_discriminant_part), then a corresponding type_declaration is nevertheless allowed to have discriminants, either explicitly, or inherited via derivation.

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 8: [AI05-0151-1]

- as the subtype_mark in an access_definition.

by:

- as the subtype_mark in an access_definition;
- as the subtype_mark defining the subtype of a parameter or result in a profile occurring within a basic_declaration.

Replace paragraph 8.2: [AI05-0151-1]

- as the subtype_mark defining the subtype of a parameter in a formal_part;

by:

- as the subtype_mark defining the subtype of a parameter in the profile for a subprogram_body, entry_body, or accept_statement;

Delete paragraph 9.1: [AI05-0151-1]

If such a name occurs within the declaration list containing the completion of the incomplete view, it may also be used:

Delete paragraph 9.2: [AI05-0098-1; AI05-0151-1]

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

Replace paragraph 10: [AI05-0151-1]

A prefix that denotes an object shall not be of an incomplete view.

by:

A prefix that denotes an object shall not be of an incomplete view. An actual parameter in a call shall not be of an untagged incomplete view. The result object of a function call shall not be of an incomplete view. A prefix shall not denote a subprogram having a formal parameter of an untagged incomplete view, nor a return type that is an incomplete view.

Insert after paragraph 13: [AI05-0151-1]

85 Within a declarative_part, an incomplete_type_declaration and a corresponding full_type_declaration cannot be separated by an intervening body. This is because a type has to be completely defined before it is frozen, and a body freezes all types declared prior to it in the same declarative_part (see 13.14).

the new paragraph:

86 A name that denotes an object of an incomplete view is defined to be of a limited type. Hence, the target of an assignment statement shall not be of an incomplete view.

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);

Insert after paragraph 13.1: [AI05-0142-4; AI05-0148-1]

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

the new paragraphs:

- The accessibility level of the type of a stand-alone object of an anonymous access-to-object type is the same as the accessibility level of the type of the access value most recently assigned to the object, but is never deeper than that of the declaration of the stand-alone object.
- The accessibility level of an explicitly aliased (see 6.1) formal parameter in a function body is determined by the point of call (which is the same level that the return object ultimately will have).

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; AI05-0066-1]

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; AI05-0142-4; AI05-0148-1]

- 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.
- The statically deeper relationship does not apply to the accessibility level of the type of a stand-alone object of an anonymous access-to-object type; that is, such an accessibility level is not considered to be statically deeper, nor statically shallower, than any other.
- Inside a return statement that applies to a function *F*, when determining whether the accessibility level of an explicitly aliased parameter of *F* is statically deeper than the level of the return object of *F*, the level of the return object is presumed to be the same as that of the level of the explicitly aliased parameter; for statically comparing with the level of other entities, the level of the return object of *F* is presumed to be the same as that of the master that elaborated the function body of *F*.

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

by:

attribute_designator ::=
identifier[(*static*_expression)]
| Access | Delta | Digits | Mod

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 15: [AI05-0147-1]

- For a parenthesized expression, the applicable index constraint is that, if any, defined for the expression.

by:

- For a parenthesized expression, the applicable index constraint is that, if any, defined for the expression;
- For a conditional_expression, the applicable index constraint for each *dependent*_expression is that, if any, defined for the conditional_expression.

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 1: [AI05-0147-1]

An *expression* is a formula that defines the computation or retrieval of a value. In this International Standard, the term "expression" refers to a construct of the syntactic category expression or of any of the other five syntactic categories defined below.

by:

An *expression* is a formula that defines the computation or retrieval of a value. In this International Standard, the term "expression" refers to a construct of the syntactic category expression or of any of the following categories: relation, simple_expression, term, factor, primary, conditional_expression.

Replace paragraph 7: [AI05-0003-1; AI05-0176-1]

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

by:

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

Insert after paragraph 9.7: [AI05-0123-1]

- When both are access-to-subprogram types, the designated profiles shall be subtype conformant.

the new paragraph:

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

Replace paragraph 14: [AI05-0123-1]

For a type extension, predefined equality is defined in terms of the primitive [(possibly user-defined)] equals operator of the parent type and of any tagged components of the extension part, and predefined equality for any other components not inherited from the parent type.

by:

For a type extension, predefined equality is defined in terms of the primitive [(possibly user-defined)] equals operator for the parent type and for any components that have a record type in the extension part, and predefined equality for any other

components not inherited from the parent type.

For a derived type whose parent is an untagged record type, predefined equality is defined in terms of the primitive (possibly user-defined) equals operator of the parent type.

Replace paragraph 15: [AI05-0123-1]

For a private type, if its full type is tagged, predefined equality is defined in terms of the primitive equals operator of the full type; if the full type is untagged, predefined equality for the private type is that of its full type.

by:

For a private type, if its full type is a record type, predefined equality is defined in terms of the primitive equals operator of the full type; otherwise, predefined equality for the private type is that of its full type.

Replace paragraph 24: [AI05-0123-1]

- Otherwise, the result is defined in terms of the primitive equals operator for any matching tagged components, and the predefined equals for any matching untagged components.

by:

- Otherwise, the result is defined in terms of the primitive equals operator for any matching components that are records, and the predefined equals for any other matching components.

If the primitive equals operator for an untagged record type is abstract, then Program_Error is raised at the point of any (implicit) call to that abstract subprogram.

Replace paragraph 30.2: [AI05-0149-1]

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

by:

- if the tested type is an access type and the named subtype excludes null, the value of the simple_expression is not null;
- if the tested type is a general access-to-object type, the type of the simple_expression is convertible to the tested type and its accessibility level is no deeper than that of the tested type; further, if the designated type is tagged and the simple_expression is non-null, the tag of the object designated by the value of the simple_expression is covered by the designated type of the tested type.

4.5.5 Multiplying Operators

Replace paragraph 19.2: [AI05-0020-1; AI05-0209-1]

- 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 or incomplete 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.

4.5.7 If Expressions

Insert new clause: [AI05-0147-1; AI05-0188-1]

A conditional_expression selects for evaluation at most one of the enclosed *dependent*_expressions, depending on some form of condition. One kind of conditional_expression is the if_expression, which selects for evaluation a *dependent*_expression

depending on the value of one or more corresponding conditions. Another kind of conditional_expression is defined in 4.5.8.

Syntax

conditional_expression ::= if_expression | case_expression

if_expression ::=
  if condition **then** *dependent*_expression
  {**elsif** condition **then** *dependent*_expression}
  [**else** *dependent*_expression]

condition ::= *boolean*_expression

Wherever the Syntax Rules allow an expression, a conditional_expression may be used in place of the expression, so long as it is immediately surrounded by parentheses.

Name Resolution Rules

A condition is expected to be of any boolean type.

If a conditional_expression is expected to be of a type *T*, then each *dependent*_expression of the conditional_expression is expected to be of type *T*. Similarly, if a conditional_expression is expected to be of some class of types, then each *dependent*_expression of the conditional_expression is subject to the same expectation. If a conditional_expression shall resolve to be of a type *T*, then each *dependent*_expression shall resolve to be of type *T*.

The possible types of a conditional_expression are further determined as follows:

- If the conditional_expression is the operand of a type conversion, the type of the conditional_expression is the target type of the conversion; otherwise
- If all of the *dependent*_expressions are of the same type, the type of the conditional_expression is that type; otherwise
- If a *dependent*_expression is of an elementary type, the type of the conditional_expression shall be covered by that type; otherwise
- If the conditional_expression is expected to be of type *T* or shall resolve to type *T*, then the conditional_expression is of type *T*.

Legality Rules

All of the *dependent*_expressions shall be convertible (see 4.6) to the type of the conditional_expression.

If the expected type of a conditional_expression is a specific tagged type, all of the *dependent*_expressions of the conditional_expression shall be dynamically tagged, or none shall be dynamically tagged; the conditional_expression is dynamically tagged if all of the *dependent*_expressions are dynamically tagged, is tag-indeterminate if all of the *dependent*_expressions are tag-indeterminate, and is statically tagged otherwise.

If there is no **else** *dependent*_expression, all of the *dependent*_expressions of the if_expression shall be of a boolean type.

Dynamic Semantics

For the execution of an if_expression, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final **else** as **elsif** True **then**), until one evaluates to True or all conditions are evaluated and yield False. If a condition evaluates to True, the associated *dependent*_expression is evaluated, converted to the type of the if_expression, and resulting value is the value of the if_expression. Otherwise, the value of the if_expression is True.

4.5.8 Case Expressions

Insert new clause: [AI05-0188-1]

One kind of conditional_expression is the case_expression, which selects for evaluation one of a number of alternative expressions; the chosen alternative is defined by the value of a *selector*_expression.

Syntax

case_expression ::=
  case *selector*_expression **is**
   case_expression_alternative {,
   case_expression_alternative}

case_expression_alternative ::=
  when discrete_choice_list =>
    *dependent*_expression

Name Resolution Rules

Name Resolution Rules for the type of the case_expression as a whole and the types of the *selector*_expressions are given in

4.5.7.

The expected type for the *selector_*expression and the discrete_choices are as for case statements (see 5.4).

Legality Rules

The expressions and discrete_ranges given as discrete_choices of a case_expression shall be static. A discrete_choice **others**, if present, shall appear alone and in the last discrete_choice_list.

The possible values of the *selector_*expression shall be covered as for case statements (see 5.4).

As for case statements, two distinct discrete_choices of a case_expression shall not cover the same value.

In addition, Legality Rules that apply to all conditional_expressions (see 4.5.7) apply to case_expressions.

Dynamic Semantics

For the execution of a case_expression, the *selector_*expression is first evaluated.

If the value of the *selector_*expression is covered by the discrete_choice_list of some case_expression_alternative, then the *dependent_*expression of the case_expression_alternative is evaluated, converted to the type of the case_expression, and the resulting value is the value of the case_expression.

Otherwise (the value is not covered by any discrete_choice_list, perhaps due to being outside the base range), Constraint_Error is raised.

4.5.9 Quantified Expressions

Insert new clause: [AI05-0176-1]

Syntax

quantified_expression ::= **for** quantifier loop_parameter_specification | predicate
quantifier ::= **all** | **some**
predicate ::= *boolean_*expression

Wherever the Syntax Rules allow an expression, a quantified_expression may be used in place of the expression, so long as it is immediately surrounded by parentheses.

Name Resolution Rules

The expected type of a quantified_expression is any Boolean type. The predicate in a quantified_expression is expected to be of the same type.

Dynamic Semantics

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

The value of the quantified_expression is determined as follows:

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

Examples

The postcondition for a sorting routine on an array A with an index subtype T can be written:

```
Post => (for all I in A'First .. T'Pred(A'Last) | A (I) <= A (T'Succ (I)))
```

The assertion that a positive number is composite (as opposed to prime) can be written:

```
pragma Assert (for some X in 2 .. N / 2 | N mod X = 0);
```

4.6 Type Conversions

Replace paragraph 24.17: [AI05-0148-1]

- The accessibility level of the operand type shall not be statically deeper than that of the target type. In addition to

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

by:

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

Replace paragraph 48: [AI05-0148-1]

- For an access-to-object type, a check is made that the accessibility level of the operand type is not deeper than that of the target type.

by:

- For an access-to-object type, a check is made that the accessibility level of the operand type is not deeper than that of the target type, unless the target is a stand-alone object of an anonymous access type. If the target is such a stand-alone object, a check is made that the accessibility level of the operand type is not deeper than that of the declaration of the stand-alone object; if the check suceeds, the accessibility level of the target type becomes that of the operand type.

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.

the new paragraph:

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; AI05-0157-1]

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.

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 Static Expressions and Static Subtypes

Insert after paragraph 12: [AI05-0147-1; AI05-0188-1]

- a short-circuit control form both of whose relations are static expressions;

the new paragraph:

- a conditional_expression all of whose conditions, *selector*_expressions, and *dependent*_expressions are static expressions;

Replace paragraph 33: [AI05-0147-1; AI05-0188-1]

A static expression is evaluated at compile time except when it is part of the right operand of a static short-circuit control form whose value is determined by its left operand. This evaluation is performed exactly, without performing Overflow_Checks. For a static expression that is evaluated:

by:

An expression is *statically unevaluated* if it is part of:

- the right operand of a static short-circuit control form whose value is determined by its left operand; or
- *dependent*_expression of an if_expression whose associated condition is static and equals False; or
- a condition or *dependent*_expression of an if_expression where the condition corresponding to at least one preceding *dependent*_expression of the if_expression is static and equals True; or
- a *dependent*_expression of a case_expression whose expression is static and not covered by the corresponding discrete_choice_list.

A static expression is evaluated at compile time except when it is statically unevaluated. The compile-time evaluation of a static expression is performed exactly, without performing Overflow_Checks. For a static expression that is evaluated:

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

5.1 Simple and Compound Statements - Sequences of Statements

Replace paragraph 2: [AI05-0179-1]

```
sequence_of_statements ::= statement {statement}
```

by:

```
sequence_of_statements ::= statement {statement} {label}
```

Insert after paragraph 12: [AI05-0179-1]

For each statement_identifier, there is an implicit declaration (with the specified identifier) at the end of the declarative_part of the innermost block_statement or body that encloses the statement_identifier. The implicit declarations occur in the same order as the statement_identifiers occur in the source text. If a usage name denotes such an implicit declaration, the entity it denotes is the label, loop_statement, or block_statement with the given statement_identifier.

the new paragraph:

If one or more labels end a sequence_of_statements, an implicit null_statement follows the labels before any following constructs.

5.3 If Statements

Delete paragraph 3: [AI05-0147-1]

condition ::= *boolean_*expression

Delete paragraph 4: [AI05-0147-1]

A condition is expected to be of any boolean type.

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

6.1 Subprogram Declarations

Replace paragraph 2: [AI05-0183-1]

subprogram_declaration ::=
  [overriding_indicator]
  subprogram_specification;

by:

subprogram_declaration ::=
  [overriding_indicator]
  subprogram_specification
    [aspect_specification];

Replace paragraph 15: [AI05-0142-4]

parameter_specification ::=
  defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]
 | defining_identifier_list : access_definition [:= default_expression]

by:

parameter_specification ::=
  defining_identifier_list : [**aliased**] mode [null_exclusion] subtype_mark [:= default_expression]
 | defining_identifier_list : access_definition [:= default_expression]

Replace paragraph 18: [AI05-0143-1]

The *parameter mode* of a formal parameter conveys the direction of information transfer with the actual parameter: **in**, **in out**, or **out**. Mode **in** is the default, and is the mode of a parameter defined by an access_definition. The formal parameters of a function, if any, shall have the mode **in**.

by:

The *parameter mode* of a formal parameter conveys the direction of information transfer with the actual parameter: **in**, **in out**, or **out**. Mode **in** is the default, and is the mode of a parameter defined by an access_definition.

Insert after paragraph 23: [AI05-0142-4]

The nominal subtype of a formal parameter is the subtype denoted by the subtype_mark, or defined by the access_definition, in the parameter_specification. The nominal subtype of a function result is the subtype denoted by the subtype_mark, or defined by the access_definition, in the parameter_and_result_profile.

the new paragraph:

An *explicitly aliased parameter* is a formal parameter whose parameter_specification includes the reserved word **aliased**.

Replace paragraph 27.1: [AI05-0164-1]

- For any access parameters of an access-to-subprogram type, the subtypes of the profile of the parameter type.

by:

- For any access parameters of an access-to-subprogram type, the subtypes of the designated profile of the parameter type.

Replace paragraph 28.2: [AI05-0164-1]

- For any access result type of an access-to-subprogram type, the subtypes of the profile of the result type.

by:

- For any access result type of an access-to-subprogram type, the subtypes of the designated profile of the result type.

6.2 Formal Parameter Modes

Replace paragraph 3: [AI05-0142-4]

A type is a *by-copy* type if it is an elementary type, or if it is a descendant of a private type whose full type is a by-copy type. A parameter of a by-copy type is passed by copy.

by:

A type is a *by-copy* type if it is an elementary type, or if it is a descendant of a private type whose full type is a by-copy type. A

parameter of a by-copy type that is not explicitly aliased is passed by copy.

Replace paragraph 7: [AI05-0096-1]

- a nonprivate type with the reserved word **limited** in its declaration;

by:

- an explicitly limited record type;

Replace paragraph 10: [AI05-0142-4]

A parameter of a by-reference type is passed by reference. Each value of a by-reference type has an associated object. For a parenthesized expression, qualified_expression, or type_conversion, this object is the one associated with the operand.

by:

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

6.3.1 Conformance Rules

Replace paragraph 16: [AI05-0142-4; AI05-0207-1]

Two profiles are *mode conformant* if they are type-conformant, and corresponding parameters have identical modes, and, for access parameters or access result types, the designated subtypes statically match, or the designated profiles are subtype conformant.

by:

Two profiles are *mode conformant* if they are type conformant, corresponding parameters have identical modes, both or neither are explicitly aliased parameters, and, for access parameters or access result types, the designated subtypes statically match and either both or neither are access-to-constant, or the designated profiles are subtype conformant.

Replace paragraph 18: [AI05-0046-1; AI05-0132-1]

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

by:

Two profiles are *fully conformant* if they are subtype-conformant, if they have access-to-subprogram results whose designated profiles are fully conformant, and corresponding parameters have:

- the same names; and
- both or neither have null_exclusions; and
- neither have default_expressions, or they both have default_expressions that are fully conformant with one another; and
- for access-to-subprogram parameters, the designated profiles are fully conformant.

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.

Replace paragraph 6: [AI05-0102-1; AI05-0142-4; AI05-0144-2]

The type of the actual parameter associated with an access parameter shall be convertible (see 4.6) to its anonymous access type.

by:

If the formal parameter is an explicitly aliased parameter, the type of the actual parameter shall be tagged or the actual parameter shall be an aliased view of an object. Further, if the formal parameter subtype *F* is untagged:

- the subtype *F* shall statically match the nominal subtype of the actual object; or
- the subtype *F* shall be unconstrained, discriminated in its full view, and unconstrained in any partial view.

In a function call, the accessibility level of the actual object for each explicitly aliased parameter shall not be statically deeper than accessibility level of the master of the function result.

Two name*s* are *known to denote the same object* if:

- both name*s* statically denote the same stand-alone object or parameter; or
- both name*s* are selected_component*s*, their prefix*es* are known to denote the same object, and their selector_name*s* denote the same component; or
- both name*s* are dereferences (implicit or explicit) and the dereferenced name*s* are known to denote the same object; or
- both name*s* are indexed_component*s*, their prefix*es* are known to denote the same object, and each of the pairs of corresponding index values are either both static expressions with the same static value or both names which are known to denote the same object; or
- both name*s* are slice*s*, their prefix*es* are known to denote the same object, and the two slice*s* have statically matching index constraints; or
- one of the two name*s* statically denotes a renaming declaration whose renamed *object*_name is known to denote the same object as the other, the prefix of any dereference within the renamed *object*_name is not a variable, and any expression within the renamed *object*_name contains no references to variables nor calls on non-static functions.

Two name*s* are *known to refer to the same object* if:

- The two name*s* are known to denote the same object; or
- One of the name*s* is a selected_component, indexed_component, or slice and its prefix is known to refer to the same object as the other name; or
- One of the two name*s* statically denotes a renaming declaration whose renamed *object*_name is known to refer to the same object as the other name.

If a call *C* has two or more parameters of mode **in out** or **out** that are of an elementary type, then the call is legal only if:

- For each name *N* that is passed as a parameter of mode **in out** or **out** to the call *C*, there is no other name among the other parameters of mode **in out** or **out** to *C* that is known to denote the same object.

If a construct *C* has two or more direct constituents that are name*s* or expression*s* whose evaluation may occur in an arbitrary order, at least one of which contains a function call with an **in out** or **out** parameter, then the construct is legal only if:

- For each name *N* that is passed as a parameter of mode **in out** or **out** to some inner function call *C2* (not including the construct *C* itself), there is no other name anywhere within a direct constituent of the construct *C* other than the one containing *C2*, that is known to refer to the same object.

For the purposes of checking this rule:

- For an array aggregate, an expression associated with a discrete_choice_list that has two or more discrete choices, or that has a nonstatic range, is considered as two or more separate occurrences of the expression;
- For a record aggregate:
  - The expression of a record_component_association is considered to occur once for each associated component; and
  - The default_expression for each record_component_association with <> for which the associated component has a default_expression is considered part of the aggregate;
- For a call, any default_expression evaluated as part of the call is considered part of the call.

Replace paragraph 13: [AI05-0196-1]

For an access type, the formal parameter is initialized from the value of the actual, without a constraint check;

by:

For an access type, the formal parameter is initialized from the value of the actual, without checking that the value satisfies any constraint or any exclusion of the null value;

Insert after paragraph 15: [AI05-0142-4]

- For any other type, the formal parameter is uninitialized. If composite, a view conversion of the actual parameter to the nominal subtype of the formal is evaluated [(which might raise Constraint_Error)], and the actual subtype of the formal is that of the view conversion. If elementary, the actual subtype of the formal is given by its nominal

subtype.

the new paragraph:

- In a function call, for each explicitly aliased parameter, a check is made that the accessibility level of the master of the actual object is not deeper than that of the master of the function result.

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; AI05-0103-1]

- 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. The subtype defined by the subtype_indication shall statically compatible with the result subtype of the function; if the result type of the function is elementary, the two subtypes shall statically match. If the result subtype of the function is indefinite, 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; AI05-0144-2]

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 that declares the return object to be a constant 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).

6.6 Overloading of Operators

Replace paragraph 3: [AI05-0143-1]

The subprogram_specification of a unary or binary operator shall have one or two parameters, respectively. A generic function instantiation whose designator is an operator_symbol is only allowed if the specification of the generic function has the corresponding number of parameters.

by:

The subprogram_specification of a unary or binary operator shall have one or two parameters, respectively. The parameters shall be of mode **in**. A generic function instantiation whose designator is an operator_symbol is only allowed if the specification of the generic function has the corresponding number of parameters of mode **in** and no other parameters.

Replace paragraph 6: [AI05-0128-1]

A declaration of "=" whose result type is Boolean implicitly declares a declaration of "/=" that gives the complementary result.

by:

An explicit declaration of "=" whose result type is Boolean implicitly declares an operator"/=" that gives the complementary result.

6.7 Null Procedures

Replace paragraph 2: [AI05-0183-1]

```
null_procedure_declaration ::=
  [overriding_indicator]
  procedure_specification is null;
```

by:

```
null_procedure_declaration ::=
  [overriding_indicator]
  procedure_specification is null
    [aspect_specification];
```

Section 7: Packages

7.1 Package Specifications and Declarations

Replace paragraph 3: [AI05-0183-1]

package_specification ::=
package defining_program_unit_name **is**
{basic_declarative_item}
[**private**
{basic_declarative_item}]
end [[parent_unit_name.]identifier]

by:

package_specification ::=
package defining_program_unit_name
[aspect_specification] **is**
{basic_declarative_item}
[**private**
{basic_declarative_item}]
end [[parent_unit_name.]identifier]

7.3 Private Types and Private Extensions

Replace paragraph 2: [AI05-0183-1]

private_type_declaration ::=
type defining_identifier [discriminant_part] **is** [[**abstract**] **tagged**] [**limited**] **private**;

by:

private_type_declaration ::=
type defining_identifier [discriminant_part] **is** [[**abstract**] **tagged**] [**limited**] **private**
[aspect_specification];

Replace paragraph 3: [AI05-0183-1]

private_extension_declaration ::=
type defining_identifier [discriminant_part] **is**
[**abstract**] [**limited** | **synchronized**] **new** ancestor_subtype_indication
[**and** interface_list] **with private**;

by:

private_extension_declaration ::=
type defining_identifier [discriminant_part] **is**
[**abstract**] [**limited** | **synchronized**] **new** ancestor_subtype_indication
[**and** interface_list] **with private**
[aspect_specification];

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 includes 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.

7.3.1 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.

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

Replace paragraph 2.1: [AI05-0147-1]

In the following contexts, an expression of a limited type is not permitted unless it is an aggregate, a function_call, or a parenthesized expression or qualified_expression whose operand is permitted by this rule:

by:

In the following contexts, an expression of a limited type is not permitted unless it is an aggregate, a function_call, a parenthesized expression or qualified_expression whose operand is permitted by this rule, or a conditional_expression all of whose *dependent_*expressions are permitted by this rule:

Replace paragraph 3: [AI05-0178-1]

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

by:

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

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 6: [AI05-0178-1]

- a composite type with a limited component;

the new paragraph:

- an incomplete view;

Insert after paragraph 8: [AI05-0052-1; AI05-0217-1]

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 record extension with the reserved word **limited**;
- 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; AI05-0142-4]

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. If such an anonymous object is part of the actual parameter expression for an explicitly aliased parameter of a function call, the master of the object is the innermost master enclosing the evaluation of the aggregate or function call, excluding the aggregate or function call itself. Otherwise, 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.

Insert after paragraph 20: [AI05-0107-1]

- For a Finalize invoked by a transfer of control due to an abort or selection of a terminate alternative, the exception is ignored; any other finalizations due to be performed are performed.

the new paragraph:

Implementation Permissions

If the execution of an allocator propagates an exception, any parts of the allocated object that were successfully initialized may be finalized as part of the finalization of the innermost master enclosing the allocator.

Section 8: Visibility Rules

8.3 Visibility

Insert after paragraph 17: [AI05-0205-1]

- For a record type or record extension, the declaration is hidden from all visibility only until the reserved word **record**;

the new paragraph:

- For an extended_return_statement, the declaration is hidden from all visibility only until the reserved word **do** of the statement;

8.4 Use Clauses

Replace paragraph 4: [AI05-0150-1]

use_type_clause ::= **use type** subtype_mark {, subtype_mark};

by:

use_type_clause ::= **use** [**all**] **type** subtype_mark {, subtype_mark};

Replace paragraph 8: [AI05-0150-1]

For each package named in a use_package_clause whose scope encloses a place, each declaration that occurs immediately within the declarative region of the package is *potentially use-visible* at this place if the declaration is visible at this place. For each type *T* or *T*'Class named in a use_type_clause whose scope encloses a place, the declaration of each primitive operator of type *T* is potentially use-visible at this place if its declaration is visible at this place.

by:

For each package named in a use_package_clause whose scope encloses a place, each declaration that occurs immediately within the declarative region of the package is *potentially use-visible* at this place if the declaration is visible at this place. For each type *T* or *T*'Class named in a use_type_clause whose scope encloses a place, the declaration of each primitive operator of type *T* is potentially use-visible at this place if its declaration is visible at this place. If a use_type_clause whose scope encloses a place includes the reserved word **all**, then the following entities are also potentially use-visible at this place if the declaration of the entity is visible at this place:

- Each primitive subprogram of *T* including each enumeration literal (if any);
- Each subprogram that is declared immediately within the declarative region in which an ancestor type of *T* is declared and that operates on a class-wide type that covers *T*.

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.

8.5.4 Subprogram Renaming Declarations

Replace paragraph 8: [AI05-0123-1]

For a call on a renaming of a dispatching subprogram that is overridden, if the overriding occurred before the renaming, then the body executed is that of the overriding declaration, even if the overriding declaration is not visible at the place of the renaming; otherwise, the inherited or predefined subprogram is called.

by:

For a call on a renaming of a dispatching subprogram that is overridden, if the overriding occurred before the renaming, then the body executed is that of the overriding declaration, even if the overriding declaration is not visible at the place of the renaming; otherwise, the inherited or predefined subprogram is called. A corresponding rule applies to a call on a renaming of a predefined equality operator for an untagged record type.

8.6 The Context of Overload Resolution

Insert after paragraph 25: [AI05-0149-1]

- when *T* is a specific anonymous access-to-object type (see 3.10) with designated type *D*, to an access-to-object type whose designated type is *D*'Class or is covered by *D*; or

the new paragraph:

- when *T* is a named general access-to-object type (see 3.10) with designated type *D*, to an anonymous access-to-object type whose designated type covers or is covered by *D*; or

Insert after paragraph 27: [AI05-0102-1; AI05-0149-1]

When a construct is one that requires that its expected type be a *single* type in a given class, the type of the construct shall be determinable solely from the context in which the construct appears, excluding the construct itself, but using the requirement that it be in the given class. Furthermore, the context shall not be one that expects any type in some class that contains types of the given class; in particular, the construct shall not be the operand of a type_conversion.

the new paragraph:

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

Replace paragraph 29: [AI05-0149-1]

There is a *preference* for the primitive operators (and ranges) of the root numeric types *root_integer* and *root_real*. In particular, if two acceptable interpretations of a constituent of a complete context differ only in that one is for a primitive operator (or range) of the type *root_integer* or *root_real*, and the other is not, the interpretation using the primitive operator (or *range*) of the root numeric type is *preferred*.

by:

There is a *preference* for the primitive operators (and ranges) of the root numeric types *root_integer* and *root_real*. In particular, if two acceptable interpretations of a constituent of a complete context differ only in that one is for a primitive operator (or range) of the type *root_integer* or *root_real*, and the other is not, the interpretation using the primitive operator (or *range*) of the root numeric type is *preferred*. Similarly, there is a preference for the equality operators of the *universal_access* type (see 4.5.2). If two acceptable interpretations of a constituent of a complete context differ only in that one is for an equality operator of the *universal_access* type, and the other is not, the interpretation using the equality operator of the *universal_access* type is preferred.

Section 9: Tasks and Synchronization

9.1 Task Units and Task Objects

Replace paragraph 2: [AI05-0183-1]

task_type_declaration ::=
  task type defining_identifier [known_discriminant_part] [**is**
    [**new** interface_list **with**]
    task_definition];

by:

task_type_declaration ::=
  task type defining_identifier [known_discriminant_part]
      [aspect_specification] [**is**
    [**new** interface_list **with**]
    task_definition];

Replace paragraph 3: [AI05-0183-1]

single_task_declaration ::=
  task defining_identifier [**is**
    [**new** interface_list **with**]
    task_definition];

by:

single_task_declaration ::=
  task defining_identifier
      [aspect_specification] [**is**
    [**new** interface_list **with**]
    task_definition];

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 2: [AI05-0183-1]

protected_type_declaration ::=
  protected type defining_identifier [known_discriminant_part] **is**
    [**new** interface_list **with**]
    protected_definition;

by:

protected_type_declaration ::=
  protected type defining_identifier [known_discriminant_part]
      [aspect_specification] **is**
    [**new** interface_list **with**]
    protected_definition;

Replace paragraph 3: [AI05-0183-1]

single_protected_declaration ::=
  protected defining_identifier **is**
    [**new** interface_list **with**]
    protected_definition;

by:

single_protected_declaration ::=
  protected defining_identifier
      [aspect_specification] **is**
    [**new** interface_list **with**]
    protected_definition;

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_Procedure 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.

9.5.2 Entries and Accept Statements

Replace paragraph 2: [AI05-0183-1]

```
entry_declaration ::=
  [overriding_indicator]
  entry defining_identifier [(discrete_subtype_definition)] parameter_profile;
```

by:

```
entry_declaration ::=
  [overriding_indicator]
  entry defining_identifier [(discrete_subtype_definition)] parameter_profile
    [aspect_specification];
```

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 1: [AI05-0009-1]

If two different objects, including nonoverlapping parts of the same object, are *independently addressable*, they can be manipulated concurrently by two different tasks without synchronization. Normally, any two nonoverlapping objects are independently addressable. However, if packing, record layout, or Component_Size is specified for a given composite object, then it is implementation defined whether or not two nonoverlapping parts of that composite object are independently addressable.

by:

If two different objects, including nonoverlapping parts of the same object, are *independently addressable*, they can be manipulated concurrently by two different tasks without synchronization. Any two nonoverlapping objects are independently addressable if either object is specified as independently addressable (see C.6). Otherwise, two nonoverlapping objects are

independently addressable except when they are both parts of a composite object for which a non-confirming representation item is used to specify packing, record layout, Component_Size, or convention, in which case it is unspecified whether the parts are independently addressable.

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 12.2: [AI05-0129-1]

- For each nested package_declaration, a declaration of the limited view of that package, with the same defining_program_unit_name.

by:

- For each nested package_declaration directly in the visible part, a declaration of the limited view of that package, with the same defining_program_unit_name.

Replace paragraph 12.3: [AI05-0108-1; AI05-0129-1]

- 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 directly in the visible part that is not an incomplete_type_declaration, 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 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; AI05-0122-1]

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 (or the corresponding declaration for a child of a generic within an instance — see 10.1.1), if it 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 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.

10.1.5 Pragmas and Program Units

Replace paragraph 7: [AI05-0132-1]

Certain program unit pragmas are defined to be *library unit pragmas*. The name, if any, in a library unit pragma shall denote the declaration of a library unit.

by:

Certain program unit pragmas are defined to be *library unit pragmas*. If a library unit pragma applies to a program unit, the program unit shall be a library unit.

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).

Replace paragraph 18: [AI05-0219-1]

If a library unit is declared pure, then the implementation is permitted to omit a call on a library-level subprogram of the library unit if the results are not needed after the call. In addition, the implementation may omit a call on such a subprogram and simply reuse the results produced by an earlier call on the same subprogram, provided that none of the parameters nor any object accessible via access values from the parameters are of a limited type, and the addresses and values of all by-reference actual parameters, the values of all by-copy-in actual parameters, and the values of all objects accessible via access values from the parameters, are the same as they were at the earlier call. This permission applies even if the subprogram produces other side effects when called.

by:

If a library unit is declared pure, then the implementation is permitted to omit a call on a library-level subprogram of the library unit if the results are not needed after the call. In addition, the implementation may omit a call on such a subprogram and simply reuse the results produced by an earlier call on the same subprogram, provided that none of the parameters nor any object accessible via access values from the parameters have any part that is of a type whose full type is an immutably limited type, and the addresses and values of all by-reference actual parameters, the values of all by-copy-in actual parameters, and the values of all objects accessible via access values from the parameters, are the same as they were at the earlier call. This permission applies even if the subprogram produces other side effects when called.

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.

11.1 Exception Declarations

Replace paragraph 2: [AI05-0183-1]

exception_declaration ::= defining_identifier_list : **exception**;

by:

exception_declaration ::= defining_identifier_list : **exception**
[aspect_specification];

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

12.1 Generic Declarations

Replace paragraph 3: [AI05-0183-1]

generic_subprogram_declaration ::=
  generic_formal_part subprogram_specification
    [aspect_specification];

by:

generic_subprogram_declaration ::=
  generic_formal_part subprogram_specification
    [aspect_specification];

12.3 Generic Instantiation

Replace paragraph 2: [AI05-0183-1]

generic_instantiation ::=
   package defining_program_unit_name **is**
     new generic_package_name [generic_actual_part];
 | [overriding_indicator]
   procedure defining_program_unit_name **is**
     new generic_procedure_name [generic_actual_part];
 | [overriding_indicator]
   function defining_designator **is**
     new generic_function_name [generic_actual_part];

by:

generic_instantiation ::=
   package defining_program_unit_name **is**
     new generic_package_name [generic_actual_part]
       [aspect_specification];
 | [overriding_indicator]
   procedure defining_program_unit_name **is**
     new generic_procedure_name [generic_actual_part]
       [aspect_specification];
 | [overriding_indicator]
   function defining_designator **is**
     new generic_function_name [generic_actual_part]
       [aspect_specification];

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.

12.4 Formal Objects

Replace paragraph 2: [AI05-0183-1]

```
formal_object_declaration ::=
   defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression];
 | defining_identifier_list : mode access_definition [:= default_expression];
```

by:

```
formal_object_declaration ::=
   defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]
     [aspect_specification];
 | defining_identifier_list : mode access_definition [:= default_expression]
     [aspect_specification];
```

12.5 Formal Types

Replace paragraph 2: [AI05-0183-1]

```
formal_type_declaration ::=
   type defining_identifier[discriminant_part] is formal_type_definition;
```

by:

```
formal_type_declaration ::=
   type defining_identifier[discriminant_part] is formal_type_definition
     [aspect_specification];
```

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 2.1: [AI05-0183-1]

```
formal_concrete_subprogram_declaration ::=
   with subprogram_specification [is subprogram_default];
```

by:

```
formal_concrete_subprogram_declaration ::=
   with subprogram_specification [is subprogram_default]
     [aspect_specification];
```

Replace paragraph 2.2: [AI05-0183-1]

```
formal_abstract_subprogram_declaration ::=
   with  subprogram_specification is abstract [subprogram_default];
```

by:

```
formal_abstract_subprogram_declaration ::=
   with  subprogram_specification is abstract [subprogram_default]
     [aspect_specification];
```

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

Replace paragraph 2: [AI05-0183-1]

```
formal_package_declaration ::=
   with  package defining_identifier is new generic_package_name formal_package_actual_part;
```

by:

formal_package_declaration ::=
with package defining_identifier **is new** generic_package_name formal_package_actual_part
[aspect_specification];

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

Replace the title: [AI05-0183-1]

Representation and operational items can be used to specify aspects of entities. Two kinds of aspects of entities can be specified: aspects of representation and operational aspects. Representation items specify how the types and other entities of the language are to be mapped onto the underlying machine. Operational items specify other properties of entities.

by:

Representation and operational items can be used to specify aspects of entities. Two kinds of aspects of entities can be specified: aspects of representation and operational aspects. Representation items specify how the types and other entities of the language are to be mapped onto the underlying machine. Operational items specify other properties of entities. In addition to representation and operational items, aspects of entities may be specified using an aspect_specification 13.3.1), which is an optional element of certain kinds of declarations.

Replace paragraph 8: [AI05-0112-1]

A representation item *directly specifies* an *aspect of representation* of the entity denoted by the local_name, except in the case of a type-related representation item, whose local_name shall denote a first subtype, and which directly specifies an aspect of the subtype's type. A representation item that names a subtype is either *subtype-specific* (Size and Alignment clauses) or *type-related* (all others). Subtype-specific aspects may differ for different subtypes of the same type.

by:

A representation item *directly specifies* an *aspect of representation* of the entity denoted by the local_name, except in the case of a type-related representation item, whose local_name shall denote a first subtype, and which directly specifies an aspect of the subtype's type. A representation item that names a subtype is either *subtype-specific* (Size and Alignment clauses) or *type-related* (all others). Subtype-specific aspects may differ for different subtypes of the same type. Unless otherwise specified, the name of the aspect of representation specified by a representation pragma is the name of the pragma.

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 representation item shall not specify an aspect of a generic formal parameter.

Replace paragraph 15: [AI05-0009-1]

A derived type inherits each type-related aspect of representation of its parent type that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent type from the grandparent type. A derived subtype inherits each subtype-specific aspect of representation of its parent subtype that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent subtype from the grandparent subtype, but only if the parent subtype statically matches the first subtype of the parent type. An inherited aspect of representation is overridden by a subsequent representation item that specifies the same aspect of the type or subtype.

by:

A derived type inherits each type-related aspect of representation of its parent type that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent type from the grandparent type. A derived subtype inherits each subtype-specific aspect of representation of its parent subtype that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent subtype from the grandparent subtype, but only if the parent subtype statically matches the first subtype of the parent type. An inherited aspect of representation is overridden by a subsequent representation item that specifies a different value for the same aspect of the type or subtype.

13.2 Pragma Pack

Replace paragraph 9: [AI05-0009-1]

- For a packed array type, if the component subtype's Size is less than or equal to the word size, and Component_Size is not specified for the type, Component_Size should be less than or equal to the Size of the component subtype, rounded up to the nearest factor of the word size.

by:

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

13.3 Representation Attributes

Replace paragraph 5: [AI05-0183-1]

An attribute_designator is allowed in an attribute_definition_clause only if this International Standard explicitly allows it, or for an implementation-defined attribute if the implementation allows it. Each specifiable attribute constitutes an operational aspect or an aspect of representation.

by:

An attribute_designator is allowed in an attribute_definition_clause only if this International Standard explicitly allows it, or for an implementation-defined attribute if the implementation allows it. Each specifiable attribute constitutes an operational aspect or an aspect of representation; the name of the aspect is that of the attribute.

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.

Replace paragraph 13: [AI05-0009-1]

If an Address is specified, it is the programmer's responsibility to ensure that the address is valid; otherwise, program execution is erroneous.

by:

If an Address is specified, it is the programmer's responsibility to ensure that the address is valid and appropriate for the entity and its use; otherwise, program execution is erroneous.

Insert before paragraph 29: [AI05-0116-1]

The recommended level of support for the Alignment attribute for subtypes is:

the new paragraph:

For any tagged specific subtype *S*, *S*'Class'Alignment should equal *S*'Alignment.

Replace paragraph 76: [AI05-0113-1]

Implementation Requirements

In an implementation, the default external tag for each specific tagged type declared in a partition shall be distinct, so long as the type is declared outside an instance of a generic body. If the compilation unit in which a given tagged type is declared, and all compilation units on which it semantically depends, are the same in two different partitions, then the external tag for the type shall be the same in the two partitions. What it means for a compilation unit to be the same in two different partitions is implementation defined. At a minimum, if the compilation unit is not recompiled between building the two different partitions that include it, the compilation unit is considered the same in the two partitions.

by:

Dynamic Semantics

If a user-specified external tag S'External_Tag is the same as T'External_Tag for some other tagged type declared by a different declaration in the partition, Program_Error is raised by the elaboration of the attribute_definition_clause.

Implementation Requirements

In an implementation, the default external tag for each specific tagged type declared in a partition shall be distinct, so long as the type is declared outside an instance of a generic body. If the compilation unit in which a given tagged type is declared, and all compilation units on which it semantically depends, are the same in two different partitions, then the external tag for the type shall be the same in the two partitions. What it means for a compilation unit to be the same in two different partitions is implementation defined. At a minimum, if the compilation unit is not recompiled between building the two different partitions that include it, the compilation unit is considered the same in the two partitions.

Implementation Permissions

If a user-specified external tag S'External_Tag is the same as T'External_Tag for some other tagged type declared by a different declaration in the partition, the partition may be rejected.

13.3.1 Aspect Specifications

Insert new clause: [AI05-0183-1]

Certain representation or operational aspects of an entity may be specified as part of its declaration using an aspect_specification, rather than using a separate representation or operational item. The declaration with the aspect_specification is termed the *associated declaration*.

Syntax

aspect_specification ::=
  with aspect_mark [=> aspect_definition] {,
    aspect_mark [=> aspect_definition] }

aspect_mark ::= *aspect_*identifier['Class]

aspect_definition ::= name | expression

Name Resolution Rules

An aspect_mark identifies an aspect of the entity defined by the associated declaration (the *associated entity*); the aspect denotes an object, a value, an expression, a subprogram, or some other kind of entity. If the aspect_mark identifies:

- an aspect that denotes an object, the aspect_definition shall be a name. The expected type for the name is the type of the identified aspect of the associated entity.
- an aspect that is a value or an expression, the aspect_definition shall be an expression. The expected type for the expression is the type of the identified aspect of the associated entity.
- an aspect that denotes a subprogram, the aspect_definition shall be a name; the expected profile for the name is the profile required for the aspect of the associated entity.
- an aspect that denotes some other kind of entity, the aspect_definition shall be a name, and the name shall resolve to denote an entity of the appropriate kind.

The usage names in an aspect_definition are not resolved at the point of the associated declaration, but rather are resolved at the end of the immediately enclosing declaration list, or at the first freezing point of the associated entity, whichever comes first.

If the associated declaration is for a subprogram or entry, the names of the formal parameters are visible within the aspect_definition, as are certain attributes, as specified elsewhere in this International Standard for the identified aspect. If the associated declaration is a type_declaration, within the aspect_definition the names of any components are visible, and the name of the first subtype denotes the current instance of the type (see 8.6). If the associated declaration is a subtype_declaration, within the aspect_definition the name of the new subtype denotes the current instance of the (sub)type.

Rest of this text TBD

13.3.2 Preconditions and Postconditions

Insert new clause: [AI05-0145-2]

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

Pre
: This aspect specifies a precondition for a callable entity; it shall be specified by an expression, called a *precondition expression*.

Pre'Class
: This aspect specifies a precondition for a callable entity and its decendants; it shall be specified by an expression, called a *precondition expression*.

Post
: This aspect specifies a postcondition for a callable entity; it shall be specified by an expression, called a *postcondition expression*.

Post'Class
: This aspect specifies a postcondition for a callable entity and its decendants; it shall be specified by an expression, called a *precondition expression*.

Name Resolution Rules

The expected type for a precondition or postcondition expression is the predefined type Boolean.

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

For an attribute_reference with attribute_designator Old, if the attribute reference has an expected type or shall resolve to a given type, the same applies to the prefix; otherwise the prefix shall be resolved independent of context.

Legality Rules

The Pre or Post aspect shall not be specified for an abstract subprogram. Only the Pre'Class and Post'Class aspects may be specified for such a subprogram.

Static Semantics

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

For a prefix X that denotes an object of a nonlimited type, the following attribute is defined:

X'Old

Denotes the value of X at the beginning of the execution of the subprogram or entry. For each X'Old in a postcondition expression that is enabled, a constant is implicitly declared at the beginning of the subprogram or entry, of the type of X, initialized to X. The value of X'Old in the postcondition expression is the value of this constant. The type of X'Old is the type of X. These implicit declarations occur in an arbitrary order.

Use of this attribute is allowed only within a postcondition expression.

For a prefix F that denotes a function declaration, the following attribute is defined:

F'Result

Within a postcondition expression for function F, denotes the result object of the function. The type of this attribute is that of the function result except within a Post'Class postcondition expression for a function with a controlling result or with a controlling access result. For a controlling result, the type of the attribute is *T*'Class, where *T* is the function result type. For a controlling access result, the type of the attribute is an anonymous access type whose designated type is *T*'Class, where *T* is the designated type of the function result type. Use of this attribute is allowed only within a postcondition expression for F.

Dynamic Semantics

If one or more precondition expressions apply to a subprogram or entry, and the Assertion_Policy in effect at the point of the subprogram or entry declaration is Check, then upon a call of the subprogram or entry, after evaluating any actual parameters, a precondition check is performed. This begins with the evaluation of the precondition expressions that apply to the subprogram or entry. If and only if all the precondition expressions evaluate to False, Ada.Assertions.Assertion_Error is raised. The order of performing the checks is not specified, and if any of the precondition expressions evaluate to True, it is not specified whether the other precondition expressions are evaluated. It is not specified whether any check for elaboration of the subprogram body is performed before or after the precondition check. It is not specified whether in a call on a protected operation, the check is performed before or after starting the protected action. For a task or protected entry call, the check is performed prior to checking whether the entry is open.

If one or more postcondition expressions apply to a subprogram or entry, and the Assertion_Policy in effect at the point of the subprogram or entry declaration is Check, then upon successful return from a call of the subprogram or entry, prior to copying back any by-copy **in out** or **out** parameters, a postcondition check is performed. This consists of the evaluation of the postcondition expressions that apply to the subprogram or entry. If any of the the postcondition expressions evaluate to False, then Ada.Assertions.Assertion_Error is raised. The order of performing the checks is not specified, and if one of them evaluates to False, it is not specified whether the others are checked. It is not specified whether any constraint checks associated with copying back **in out** or **out** parameters are performed before or after the postcondition check.

If a precondition or postcondition check fails, the exception is raised at the point of the call. The exception cannot be handled inside the called subprogram.

For a dispatching call or a call via an access-to-subprogram value, the precondition or postcondition check performed is determined by the subprogram actually invoked. Note that for a dispatching call, if there is a Pre'Class aspect that applies to the subprogram named in the call, then if the precondition expression for that aspect evaluates to True, the precondition check for the call will succeed.

If the Assertion_Policy in effect at the point of a subprogram or entry declaration is Ignore, then no precondition or postcondition check is performed on a call on that subprogram or entry. If the Assertion_Policy in effect at the point of a subprogram or entry declaration is Check, then preconditions and postconditions are considered to be *enabled* for that subprogram or entry.

Bounded Errors

It is a bounded error to invoke a potentially blocking operation (see 9.5.1) during the evaluation of a precondition or postcondition expression. If the bounded error is detected, Program_Error is raised. If not detected, execution proceeds normally,

but if it is invoked within a protected action, it might result in deadlock or a (nested) protected action.

NOTES

A precondition is checked just before the call. If another task can change any value that the precondition expression depends on, the precondition may not hold within the subprogram or entry body.

13.3.3 Type Invariants

Insert new clause: [AI05-0146-1]

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

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

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

Name Resolution Rules

The expected type for an invariant expression is any boolean type.

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

Legality Rules

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

Static Semantics

If the Type_Invariant'Class aspect is specified for a tagged type *T*, then the invariant expression applies to all descendants of *T*.

Dynamic Semantics

If one or more invariant expressions apply to a type *T*, and the Assertion_Policy at the point of the partial view declaration for *T* is Check, then an invariant check is performed at the following places, on the specified object(s):

- After default initialization of an object of type *T*, on the new object;
- After conversion to type *T*, on the result of the conversion;
- After a call on the Read or Input stream attribute of the type *T*, on the object initialized by the stream attribute;
- Upon return from a call on any subprogram or entry that:
  - has a result of type *T*, or one or more **in out** or **out** parameters of type *T*,
  - is explicitly declared within the immediate scope of type *T* (or is a part of an instance of a generic unit declared within the immediate scope of type *T*), and
  - is visible outside the immediate scope of type *T* or overrides an operation that is visible outside the immediate scope of *T*,

  on the result object of type *T*, and on each **in out** or **out** actual parameter of type *T*.

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

The invariant checks performed on a call are determined by the subprogram or entry actually invoked, whether directly, as part of a dispatching call, or as part of a call through an access-to-subprogram value.

If the Assertion_Policy in effect at the point of a subprogram or entry declaration is Ignore, then no invariant check is performed on a call on that subprogram or entry.

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 Storage Management

Replace paragraph 16: [AI05-0107-1; AI05-0116-1]

An allocator of type T allocates storage from T's storage pool. If the storage pool is a user-defined object, then the storage is allocated by calling Allocate, passing T'Storage_Pool as the Pool parameter. The Size_In_Storage_Elements parameter indicates the number of storage elements to be allocated, and is no more than D'Max_Size_In_Storage_Elements, where D is the designated subtype. The Alignment parameter is D'Alignment. The result returned in the Storage_Address parameter is used by the allocator as the address of the allocated storage, which is a contiguous block of memory of Size_In_Storage_Elements storage elements. Any exception propagated by Allocate is propagated by the allocator.

by:

An allocator of type T allocates storage from T's storage pool. If the storage pool is a user-defined object, then the storage is allocated by calling Allocate as described below.

Insert after paragraph 21: [AI05-0107-1; AI05-0116-1; AI05-0193-1]

If Storage_Pool is specified for an access type, then if Allocate can satisfy the request, it should allocate a contiguous block of memory, and return the address of the first storage element in Storage_Address. The block should contain Size_In_Storage_Elements storage elements, and should be aligned according to Alignment. The allocated storage should not be used for any other purpose while the pool element remains in existence. If the request cannot be satisfied, then Allocate should propagate an exception (such as Storage_Error). If Allocate behaves in any other manner, then the program execution is erroneous.

the new paragraphs:

Implementation Requirements

The Allocate procedure of a user-defined storage pool object *P* may be called by the implementation only to allocate storage for a type *T* whose pool is *P* and:

- During the execution of an allocator of type *T*;
- During the execution of a return statement for a function whose result is built-in-place in the result of an allocator of type *T*;
- During the execution of an assignment operation with a target of an allocated object of type *T* with a part that has an unconstrained discriminated subtype with defaults.

For one of the calls of Allocate described above, *P* (equivalent to *T*'Storage_Pool) is passed as the Pool parameter. The Size_In_Storage_Elements parameter indicates the number of storage elements to be allocated, and is no more than *D*'Max_Size_In_Storage_Elements, where *D* is the designated subtype of *T*. The Alignment parameter is at least *D*'Alignment if *D* is a specific type, and otherwise is at least the alignment of the specific type identified by the tag of the object being created. The Alignment parameter is no more than *D*'Max_Alignment_For_Allocation. The result returned in the Storage_Address

parameter is used as the address of the allocated storage, which is a contiguous block of memory of Size_In_Storage_Elements storage elements. Any exception propagated by Allocate is propagated by the construct that contained the call.

The number of calls to Allocate needed to implement an allocator for any particular type is unspecified. The number of calls to Deallocate needed to implement an instance of Unchecked_Deallocation (see 13.11.2) for any particular object is the same as the number of Allocate calls for that object.

The Deallocate procedure of a user-defined storage pool object *P* may be called by the implementation to deallocate storage for a type *T* whose pool is *P* only at the places when an Allocate call is allowed for *P*, during the execution of an instance of Unchecked_Deallocation for *T*, or as part of the finalization of the collection of *T*. For such a call of Deallocate, *P* (equivalent to *T*'Storage_Pool) is passed as the Pool parameter. The value of the Storage_Address parameter for a call to Deallocate is the value returned in the Storage_Address parameter of the corresponding successful call to Allocate. The values of the Size_In_Storage_Elements and Alignment parameters are the same values passed to the corresponding Allocate call. Any exception propagated by Deallocate is propagated by the construct that contained the call.

13.11.1 The Max_Size_In_Storage_Elements Attribute

Replace the title: [AI05-0193-1]

The Max_Size_In_Storage_Elements Attribute

by:

Storage Allocation Attributes

Replace paragraph 1: [AI05-0193-1]

The Max_Size_In_Storage_Units attribute is useful in writing user-defined pool types.

by:

The Max_Size_In_Storage_Units and Max_Alignment_For_Allocation attributes may be useful in writing user-defined pool types.

Replace paragraph 2: [AI05-0193-1]

For every subtype S, the following attribute is defined:

by:

For every subtype S, the following attributes are defined:

Replace paragraph 3: [AI05-0193-1]

S'Max_Size_In_Storage_Elements
: Denotes the maximum value for Size_In_Storage_Elements that could be requested by the implementation via Allocate for an access type whose designated subtype is S. For a type with access discriminants, if the implementation allocates space for a coextension in the same pool as that of the object having the access discriminant, then this accounts for any calls on Allocate that could be performed to provide space for such coextensions. The value of this attribute is of type *universal_integer*.

by:

S'Max_Size_In_Storage_Elements
: Denotes the maximum value for Size_In_Storage_Elements that could be requested by the implementation via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*.

S'Max_Alignment_For_Allocation
: Denotes the maximum value for Alignment that could be requested by the implementation via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*.

For a type with access discriminants, if the implementation allocates space for a coextension in the same pool as that of the object having the access discriminant, then these attributes account for any calls on Allocate that could be performed to provide space for such coextensions.

13.11.2 Unchecked Storage Deallocation

the new paragraph:

Given an instance of Unchecked_Deallocation declared as follows:

Insert before paragraph 4: [AI05-0157-1]

Legality Rules

A call on an instance of Unchecked_Deallocation is illegal if the actual access type of the instance is a 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.

Replace paragraph 9: [AI05-0107-1]

3. Free(X), when X is not equal to **null** first performs finalization of the object designated by X (and any coextensions of the object — see 3.10.2), as described in 7.6.1. It then deallocates the storage occupied by the object designated by X (and any coextensions). If the storage pool is a user-defined object, then the storage is deallocated by calling Deallocate, passing *access_to_variable_subtype_name*'Storage_Pool as the Pool parameter. Storage_Address is the value returned in the Storage_Address parameter of the corresponding Allocate call. Size_In_Storage_Elements and Alignment are the same values passed to the corresponding Allocate call. There is one exception: if the object being freed contains tasks, the object might not be deallocated.

by:

3. Free(X), when X is not equal to **null** first performs finalization of the object designated by X (and any coextensions of the object — see 3.10.2), as described in 7.6.1. It then deallocates the storage occupied by the object designated by X (and any coextensions). If the storage pool is a user-defined object, then the storage is deallocated by calling Deallocate as described in 13.11. There is one exception: if the object being freed contains tasks, the object might not be deallocated.

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.

the new paragraph:

For a standard storage pool, Free should actually reclaim the storage.

Insert after paragraph 17: [AI05-0157-1]

A call on an instance of Unchecked_Deallocation with a non-null access value should raise Program_Error if the actual access type of the instance is a type for which the Storage_Size has been specified to be zero or is defined by the language to be zero.

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.1 The Package Standard

In paragraph 35 replace: [AI05-0181-1]

```
'¨', '©', 'ª', '«', '¬', '', '®', '¯',     --168 (16#A8#) .. 175 (16#AF#)
```

by:

```
'¨', '©', 'ª', '«', '¬', soft_hyphen, '®', '¯', --168 (16#A8#) .. 175 (16#AF#)
```

A.3.2 The Package Characters.Handling

Insert after paragraph 59: [AI05-0114-1]

— Special graphic characters

the new paragraph:

7 There are certain characters which are defined to be lower case letters by ISO 10646 and are therefore allowed in identifiers, but are not considered lower case letters by Ada.Characters.Handling.

A.3.3 The Package Characters.Latin_1

In paragraph 21 replace: [AI05-0181-1]

```
Soft_Hyphen             : constant Character := ''; --Character'Val(173)
```

by:

```
Soft_Hyphen             : constant Character := Character'Val(173);
```

A.3.5 The Package Wide_Characters.Handling

Insert new clause: [AI05-0185-1]

The package Wide_Characters.Handling provides operations for classifying Wide_Characters and case folding for Wide_Characters.

Static Semantics

The library package Wide_Characters.Handling has the following declaration:

```
package Ada.Wide_Characters.Handling is

   function Is_Control (Item : Wide_Character) return Boolean;

   function Is_Letter (Item : Wide_Character) return Boolean;

   function Is_Lower (Item : Wide_Character) return Boolean;
```

```
   function Is_Upper (Item : Wide_Character) return Boolean;

   function Is_Digit (Item : Wide_Character) return Boolean;

   function Is_Decimal_Digit (Item : Wide_Character) return Boolean
      renames Is_Digit;

   function Is_Hexadecimal_Digit (Item : Wide_Character) return Boolean;

   function Is_Alphanumeric (Item : Wide_Character) return Boolean;

   function Is_Special (Item : Wide_Character) return Boolean;

   function Is_Line_Terminator (Item : Wide_Character) return Boolean;

   function Is_Mark (Item : Wide_Character) return Boolean;

   function Is_Other (Item : Wide_Character) return Boolean;

   function Is_Punctuation (Item : Wide_Character) return Boolean;

   function Is_Space (Item : Wide_Character) return Boolean;

   function Is_Graphic (Item : Wide_Character) return Boolean;

   function To_Lower (Item : Wide_Character) return Wide_Character;
   function To_Upper (Item : Wide_Character) return Wide_Character;

   function To_Lower (Item : Wide_String) return Wide_String;
   function To_Upper (Item : Wide_String) return Wide_String;

end Ada.Wide_Characters.Handling;
```

The subprograms defined in Ada.Wide_Characters.Handling are locale independent.

```
function Is_Control (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as other_control, otherwise returns False.

```
function Is_Letter (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as letter_uppercase, letter_lowercase, letter_titlecase, letter_modifier, letter_other, or number_letter. Otherwise returns False.

```
function Is_Lower (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as letter_lowercase, otherwise returns False.

```
function Is_Upper (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as letter_uppercase, otherwise returns False.

```
function Is_Digit (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as number_decimal, otherwise returns False.

```
function Is_Hexadecimal_Digit (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as number_decimal, or is in the range 'A' .. 'F' or 'a' .. 'f', otherwise returns False.

```
function Is_Alphanumeric (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as letter_uppercase, letter_lowercase, letter_titlecase, letter_modifier, letter_other, number_letter, or number_decimal. Otherwise returns False.

```
function Is_Special (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as graphic_character, but not categorized as letter_uppercase, letter_lowercase, letter_titlecase, letter_modifier, letter_other, number_letter, or number_decimal. Otherwise returns False.

```
function Is_Line_Terminator (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as separator_line or separator_paragraph, or if Item is a conventional line terminator character (CR, LF, VT, or FF). Otherwise returns False.

```
function Is_Mark (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as mark_non_spacing or mark_spacing_combining, otherwise returns False.

```
function Is_Other (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as other_format, otherwise returns False.

```
function Is_Punctuation (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as punctuation_connector, otherwise returns False.

```
function Is_Space (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as separator_space, otherwise returns False.

```
function Is_Graphic (Item : Wide_Character) return Boolean;
```

Returns True if the Wide_Character designated by Item is categorized as graphic_character, otherwise returns False.

```
function To_Lower (Item : Wide_Character) return Wide_Character;
```

Returns the Simple Lowercase Mapping of the Wide_Character designated by Item. If the Simple Lowercase Mapping does not exist for the Wide_Character designated by Item, then the value of Item is returned.

```
function To_Lower (Item : Wide_String) return Wide_String;
```

Returns the result of applying the To_Lower Wide_Character to Wide_Character conversion to each element of the Wide_String designated by Item. The result is the null Wide_String if the value of the formal parameter is the null Wide_String.

```
function To_Upper (Item : Wide_Character) return Wide_Character;
```

Returns the Simple Uppercase Mapping of the Wide_Character designated by Item. If the Simple Uppercase Mapping does not exist for the Wide_Character designated by Item, then the value of Item is returned.

```
function To_Upper (Item : Wide_String) return Wide_String;
```

Returns the result of applying the To_Upper Wide_Character to Wide_Character conversion to each element of the Wide_String designated by Item. The result is the null Wide_String if the value of the formal parameter is the null Wide_String.

A.3.6 The Package Wide_Wide_Characters.Handling

Insert new clause: [AI05-0185-1]

The package Wide_Wide_Characters.Handling has the same contents as Wide_Character.Handling except that each occurrence of Wide_Character is replaced by Wide_Wide_Character, and each occurrence of Wide_String is replaced by Wide_Wide_String.

A.4.3 Fixed-Length String Handling

Insert before paragraph 16: [AI05-0031-1]

```
procedure Find_Token (Source : in String;
                      Set    : in Maps.Character_Set;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

the new paragraph:

```
procedure Find_Token (Source : in String;
                      Set    : in Maps.Character_Set;
                      From   : in Positive;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

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.

Insert before paragraph 67: [AI05-0031-1]

```
procedure Find_Token (Source : in String;
                      Set    : in Maps.Character_Set;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

the new paragraphs:

```
procedure Find_Token (Source : in String;
                      Set    : in Maps.Character_Set;
                      From   : in Positive;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

If Source is not the null string and From is not in Source'Range, then Index_Error is raised. Otherwise, First is set to the index of the first character in Source(From..Source'Last) that satisfies the Test condition. Last is set to the largest index such that all characters in Source(First..Last) satisfy the Test condition. If no characters in Source(From..Source'Last) satisfy the Test condition, First is set to From, and Last is set to 0.

Replace paragraph 68: [AI05-0031-1]

Find_Token returns in First and Last the indices of the beginning and end of the first slice of Source all of whose elements satisfy the Test condition, and such that the elements (if any) immediately before and after the slice do not satisfy the Test condition. If no such slice exists, then the value returned for Last is zero, and the value returned for First is Source'First; however, if Source'First is not in Positive then Constraint_Error is raised.

by:

Equivalent to Find_Token (Source, Set, Source'First, Test, First, Last).

A.4.4 Bounded-Length String Handling

Insert before paragraph 51: [AI05-0031-1]

```
procedure Find_Token (Source : in Bounded_String;
                      Set    : in Maps.Character_Set;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

the new paragraph:

```
procedure Find_Token (Source : in Bounded_String;
                      Set    : in Maps.Character_Set;
                      From   : in Positive;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

A.4.5 Unbounded-Length String Handling

Insert before paragraph 46: [AI05-0031-1]

```
procedure Find_Token (Source : in Unbounded_String;
                      Set    : in Maps.Character_Set;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

the new paragraph:

```
procedure Find_Token (Source : in Unbounded_String;
                      Set    : in Maps.Character_Set;
                      From   : in Positive;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

A.4.6 String-Handling Sets and Mappings

Insert after paragraph 7: [AI05-0114-1]

Each of these constants represents a correspondingly named set of characters or character mapping in Characters.Handling (see A.3.2).

the new paragraph:

NOTES

12 There are certain characters which are defined to be lower case letters by ISO 10646 and are therefore allowed in identifiers, but are not considered lower case letters by Ada.Strings.Maps.Constants.

A.4.9 String Hashing

Replace paragraph 8: [AI05-0001-1]

Strings.Bounded.Hash is equivalent to the function call Strings.Hash (Bounded.To_String (Key));

by:

Equivalent to Strings.Hash (Bounded.To_String (Key));

Replace paragraph 11: [AI05-0001-1]

Strings.Unbounded.Hash is equivalent to the function call Strings.Hash (To_String (Key));

by:

Equivalent to Strings.Hash (To_String (Key));

The library function Strings.Hash_Case_Insensitive has the following declaration:

```
with Ada.Containers;
function Ada.Strings.Hash_Case_Insensitive (Key : String)
   return Containers.Hash_Type;
pragma Pure(Hash_Case_Insensitive);
```

Returns an implementation-defined value which is a function of the value of Key, converted to lower case. If A and B are strings such that Strings.Equal_Case_Insensitive (A, B) (see A.4.10) is True, then Hash_Case_Insensitive(A) equals Hash_Case_Insensitive(B).

The library function Strings.Fixed.Hash_Case_Insensitive has the following declaration:

```
with Ada.Containers, Ada.Strings.Hash_Case_Insensitive;
function Ada.Strings.Fixed.Hash_Case_Insensitive (Key : String)
   return Containers.Hash_Type renames Ada.Strings.Hash_Case_Insensitive;
pragma Pure(Hash_Case_Insensitive);
```

The generic library function Strings.Bounded.Hash_Case_Insensitive has the following declaration:

```
with Ada.Containers;
generic
   with package Bounded is
          new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Hash_Case_Insensitive
   (Key : Bounded.Bounded_String) return Containers.Hash_Type;
pragma Preelaborate(Hash_Case_Insensitive);
```

Equivalent to Strings.Hash_Case_Insensitive (Bounded.To_String (Key));

The library function Strings.Unbounded.Hash_Case_Insensitive has the following declaration:

```
with Ada.Containers;
function Ada.Strings.Unbounded.Hash_Case_Insensitive
   (Key : Unbounded_String) return Containers.Hash_Type;
pragma Preelaborate(Hash_Case_Insensitive);
```

Equivalent to Strings.Hash_Case_Insensitive (To_String (Key));

A.4.10 String Comparison

Insert new clause: [AI05-0001-1]

Static Semantics

The library function Strings.Equal_Case_Insensitive has the following declaration:

```
function Ada.Strings.Equal_Case_Insensitive (Left, Right : String)
   return Boolean;
pragma Pure(Equal_Case_Insensitive);
```

Compares strings Left and Right, converted to lower case, for equality.

The library function Strings.Fixed.Equal_Case_Insensitive has the following declaration:

```
with Ada.Containers, Ada.Strings.Equal_Case_Insensitive;
function Ada.Strings.Fixed.Equal_Case_Insensitive
   (Left, Right : String) return Boolean
      renames Ada.Strings.Equal_Case_Insensitive;
pragma Pure(Equal_Case_Insensitive);
```

The generic library function Strings.Bounded.Equal_Case_Insensitive has the following declaration:

```
with Ada.Containers;
generic
   with package Bounded is
          new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Equal_Case_Insensitive
   (Left, Right : Bounded.Bounded_String) return Boolean;
pragma Preelaborate(Equal_Case_Insensitive);
```

Equivalent to Strings.Equal_Case_Insensitive (Bounded.To_String (Key));

The library function Strings.Unbounded.Equal_Case_Insensitive has the following declaration:

```
with Ada.Containers;
function Ada.Strings.Unbounded.Equal_Case_Insensitive
   (Left, Right : Unbounded_String) return Boolean;
pragma Preelaborate(Equal_Case_Insensitive);
```

Equivalent to Strings.Equal_Case_Insensitive (To_String (Key));

The library function Strings.Less_Case_Insensitive has the following declaration:

```
function Ada.Strings.Less_Case_Insensitive (Left, Right : String)
   return Boolean;
pragma Pure(Less_Case_Insensitive);
```

Performs a lexicographic comparison of strings Left and Right, converted to lower case.

The library function Strings.Fixed.Less_Case_Insensitive has the following declaration:

```
with Ada.Containers, Ada.Strings.Less_Case_Insensitive;
function Ada.Strings.Fixed.Less_Case_Insensitive
   (Left, Right : String) return Boolean
      renames Ada.Strings.Less_Case_Insensitive;
pragma Pure(Less_Case_Insensitive);
```

The generic library function Strings.Bounded.Less_Case_Insensitive has the following declaration:

```
with Ada.Containers;
generic
   with package Bounded is
          new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Less_Case_Insensitive
  (Left, Right : Bounded.Bounded_String) return Boolean;
pragma Preelaborate(Less_Case_Insensitive);
```

Equivalent to Strings.Less_Case_Insensitive (Bounded.To_String (Key));

The library function Strings.Unbounded.Less_Case_Insensitive has the following declaration:

```
with Ada.Containers;
function Ada.Strings.Unbounded.Less_Case_Insensitive
  (Left, Right : Unbounded_String) return Boolean;
pragma Preelaborate(Less_Case_Insensitive);
```

Equivalent to Strings.Less_Case_Insensitive (To_String (Key));

A.4.11 String Encoding

Insert new clause: [AI05-0137-2]

Facilities for encoding, decoding, and converting strings in various character encoding schemes are provided by packages Strings.UTF_Encoding, Strings.UTF_Encoding.Conversions, Strings.UTF_Encoding.Strings, Strings.UTF_Encoding.Wide_Strings, and Strings.UTF_Encoding.Wide_Wide_Strings.

Static Semantics

The encoding library packages have the following declarations:

```
package Ada.Strings.UTF_Encoding is
   pragma Pure (UTF_Encoding);

   -- Declarations common to the string encoding packages
   type Encoding_Scheme is (UTF_8, UTF_16BE, UTF_16LE);

   subtype UTF_String is String;

   subtype UTF_8_String is String;

   subtype UTF_16_Wide_String is Wide_String;

   Encoding_Error : exception;

   BOM_8    : constant UTF_8_String :=
               Character'Val (16#EF#) &
               Character'Val (16#BB#) &
               Character'Val (16#BF#);

   BOM_16BE : constant UTF_String :=
               Character'Val (16#FE#) &
               Character'Val (16#FF#);

   BOM_16LE : constant UTF_String :=
               Character'Val (16#FF#) &
               Character'Val (16#FE#);

   BOM_16   : constant UTF_16_Wide_String :=
               (1 => Wide_Character'Val (16#FEFF#));

   function Encoding (Item    : UTF_String;
                      Default : Encoding_Scheme := UTF_8) return Encoding_Scheme;

end Ada.Strings.UTF_Encoding;

package Ada.Strings.UTF_Encoding.Conversions is
   pragma Pure (Conversions);

   -- Conversions between various encoding schemes
   function Convert (Item          : UTF_String;
                     Input_Scheme  : Encoding_Scheme;
                     Output_Scheme : Encoding_Scheme;
                     Output_BOM    : Boolean := False) return UTF_String;

   function Convert (Item          : UTF_String;
                     Input_Scheme  : Encoding_Scheme;
                     Output_BOM    : Boolean := False) return UTF_16_Wide_String;

   function Convert (Item          : UTF_8_String;
                     Output_BOM    : Boolean := False) return UTF_16_Wide_String;

   function Convert (Item          : UTF_16_Wide_String;
                     Output_Scheme : Encoding_Scheme;
                     Output_BOM    : Boolean := False) return UTF_String;

   function Convert (Item          : UTF_16_Wide_String;
                     Output_BOM    : Boolean := False) return UTF_8_String;

end Ada.Strings.UTF_Encoding.Conversions;
```

```
package Ada.Strings.UTF_Encoding.Strings is
   pragma Pure (Strings);

   -- Encoding / decoding between String and various encoding schemes
   function Encode (Item          : String;
                    Output_Scheme : Encoding_Scheme;
                    Output_BOM    : Boolean  := False) return UTF_String;

   function Encode (Item       : String;
                    Output_BOM : Boolean  := False) return UTF_8_String;

   function Encode (Item       : String;
                    Output_BOM : Boolean  := False) return UTF_16_Wide_String;

   function Decode (Item         : UTF_String;
                    Input_Scheme : Encoding_Scheme) return String;

   function Decode (Item : UTF_8_String) return String;

   function Decode (Item : UTF_16_Wide_String) return String;

end Ada.Strings.UTF_Encoding.Strings;

package Ada.Strings.UTF_Encoding.Wide_Strings is
   pragma Pure (Wide_Strings);

   -- Encoding / decoding between Wide_String and various encoding schemes
   function Encode (Item          : Wide_String;
                    Output_Scheme : Encoding_Scheme;
                    Output_BOM    : Boolean  := False) return UTF_String;

   function Encode (Item       : Wide_String;
                    Output_BOM : Boolean  := False) return UTF_8_String;

   function Encode (Item       : Wide_String;
                    Output_BOM : Boolean  := False) return UTF_16_Wide_String;

   function Decode (Item         : UTF_String;
                    Input_Scheme : Encoding_Scheme) return Wide_String;

   function Decode (Item : UTF_8_String) return Wide_String;

   function Decode (Item : UTF_16_Wide_String) return Wide_String;

end Ada.Strings.UTF_Encoding.Wide_Strings;

package Ada.Strings.UTF_Encoding.Wide_Wide_Strings is
   pragma Pure (Wide_Wide_Strings);

   -- Encoding / decoding between Wide_Wide_String and various encoding schemes
   function Encode (Item          : Wide_Wide_String;
                    Output_Scheme : Encoding_Scheme;
                    Output_BOM    : Boolean  := False) return UTF_String;

   function Encode (Item       : Wide_Wide_String;
                    Output_BOM : Boolean  := False) return UTF_8_String;

   function Encode (Item       : Wide_Wide_String;
                    Output_BOM : Boolean  := False) return UTF_16_Wide_String;

   function Decode (Item         : UTF_String;
                    Input_Scheme : Encoding_Scheme) return Wide_Wide_String;

   function Decode (Item : UTF_8_String) return Wide_Wide_String;

   function Decode (Item : UTF_16_Wide_String) return Wide_Wide_String;

end Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
```

The type Encoding_Scheme defines encoding schemes. UTF_8 corresponds to the UTF-8 encoding scheme defined by Annex D of ISO/IEC 106046. UTF_16BE corresponds to the UTF-16 encoding scheme defined by Annex C of ISO/IEC 106046 stored in 8 bits, big endian; and UTF_16LE corresponds to the UTF-16 encoding scheme on 8 bits, little endian.

The subtype UTF_String is used to represent a String of 8-bit values containing a sequence of values encoded in one of three ways (UTF-8, UTF-16BE, or UTF-16LE). The subtype UTF_8_String is used to represent a String of 8-bit values containing a sequence of values encoded in UTF-8. The subtype UTF_16_Wide_String is used to represent a Wide_String of 16-bit values containing a sequence of values encoded in UTF-16.

The BOM_8, BOM_16BE, BOM_16LE and BOM_16 constants correspond to values used at the start of a string to indicate the encoding.

For all Convert and Decode functions, an initial BOM in the input that matches the expected encoding scheme is ignored, and a different initial BOM causes Encoding_Error to be propagated.

For all Convert and Encode functions, a BOM is included at the start of the output string if the Output_BOM parameter is set to True.

The exception Encoding_Error is also propagated in the following situations:

- By a Decode function when an UTF encoded string contains an invalid encoding sequence.
- By a Decode function when the expected encoding is UTF-16BE or UTF-16LE and the input string has an odd length.
- By a Decode function yielding a String when the decoding of a sequence results in a code-point whose value exceeds 16#FF#.
- By a Decode function yielding a Wide_String when the decoding of a sequence results in a code-point whose value exceeds 16#FFFF#.
- By an Encode function taking a Wide_String as input when an invalid character appears in the input. In particular the characters whose position is in the range 16#D800# .. 16#DFFF# are invalid because they conflict with UTF-16 surrogate encodings, and the characters whose position is 16#FFFE# or 16#FFFF# are also invalid because they conflict with BOM codes.

Each of the Convert and Encode functions returns a String (respectively Wide_String) value whose characters have position values that correspond to the encoding of the Item parameter according to the encoding scheme required by the function or specified by its Output_Scheme parameter. For UTF_8, no overlong encoding is returned. The lower bound of the returned string is 1.

Each of the Decode functions takes a String (respectively Wide_String) Item parameter which is assumed to contain characters whose position values correspond to a valid encoding sequence according to the encoding scheme required by the function or specified by its Input_Scheme parameter, and returns the corresponding String, Wide_String, or Wide_Wide_String value. The lower bound of the returned string is 1.

```
function Encoding (Item    : UTF_String;
                   Default : Encoding_Scheme := UTF_8) return Encoding_Scheme;
```

Inspects a UTF_String value to determine whether it starts with a BOM for UTF-8, UTF-16BE, or UTF_16LE. If so, returns the scheme corresponding to the BOM; returns the value of Default otherwise.

```
function Convert (Item          : UTF_String;
                  Input_Scheme  : Encoding_Scheme;
                  Output_Scheme : Encoding_Scheme;
                  Output_BOM    : Boolean := False) return UTF_String;
```

Converts from input encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme, and generates an output encoded in one of these three schemes as specified by Output_Scheme.

```
function Convert (Item          : UTF_String;
                  Input_Scheme  : Encoding_Scheme;
                  Output_BOM    : Boolean := False) return UTF_16_Wide_String;
```

Converts from input encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme, and generates an output encoded in UTF-16.

```
function Convert (Item          : UTF_8_String;
                  Output_BOM    : Boolean := False) return UTF_16_Wide_String;
```

Converts from input encoded in UTF-8 and generates an output encoded in UTF-16.

```
function Convert (Item          : UTF_16_Wide_String;
                  Output_Scheme : Encoding_Scheme;
```

```
                  Output_BOM    : Boolean := False) return UTF_String;
```

Converts from input encoded in UTF-16 and generates an output encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Output_Scheme.

```
function Convert (Item          : UTF_16_Wide_String;
                  Output_BOM    : Boolean := False) return UTF_8_String;
```

Converts from input encoded in UTF-16 and generates an output encoded in UTF-8.

```
function Encode (Item          : String;
                 Output_Scheme : Encoding_Scheme;
                 Output_BOM    : Boolean  := False) return UTF_String;
```

Encodes from String input, and generates an output encoded in UTF-8, UTF-16LE or UTF-16BE encoding as specified by Output_Scheme.

```
function Encode (Item       : String;
                 Output_BOM : Boolean  := False) return UTF_8_String;
```

Encodes from String input, and generates an output encoded in UTF-8 encoding.

```
function Encode (Item       : String;
                 Output_BOM : Boolean  := False) return UTF_16_Wide_String;
```

Encodes from String input, and generates an output encoded in UTF_16 encoding.

```
function Decode (Item         : UTF_String;
                 Input_Scheme : Encoding_Scheme) return String;
```

Decodes from input encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme, and returns the corresponding String value.

```
function Decode (Item : UTF_8_String) return String;
```

Decodes from input encoded in UTF-8, and returns the corresponding String value.

```
function Decode (Item : UTF_16_Wide_String) return String;
```

Decodes from input encoded in UTF-16, and returns the corresponding String value.

```
function Encode (Item          : Wide_String;
                 Output_Scheme : Encoding_Scheme;
                 Output_BOM    : Boolean  := False) return UTF_String;
```

Encodes from Wide_String input, and generates an output encoded in UTF-8, UTF-16LE or UTF-16BE encoding as specified by Output_Scheme.

```
function Encode (Item       : Wide_String;
                 Output_BOM : Boolean  := False) return UTF_8_String;
```

Encodes from String input, and generates an output encoded in UTF-8 encoding.

```
function Encode (Item       : Wide_String;
                 Output_BOM : Boolean  := False) return UTF_16_Wide_String;
```

Encodes from Wide_String input, and generates an output encoded in UTF_16 encoding.

```
function Decode (Item         : UTF_String;
                 Input_Scheme : Encoding_Scheme) return Wide_String;
```

Decodes from input encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme, and returns the corresponding Wide_String value.

```
function Decode (Item : UTF_8_String) return Wide_String;
```

Decodes from input encoded in UTF-8, and returns the corresponding Wide_String value.

```
function Decode (Item : UTF_16_Wide_String) return Wide_String;
```

Decodes from input encoded in UTF-16, and returns the corresponding Wide_String value.

```
function Encode (Item          : Wide_Wide_String;
                 Output_Scheme : Encoding_Scheme;
                 Output_BOM    : Boolean  := False) return UTF_String;
```

Encodes from Wide_Wide_String input, and generates an output encoded in UTF-8, UTF-16LE or UTF-16BE encoding as specified by Output_Scheme.

```
function Encode (Item       : Wide_Wide_String;
                 Output_BOM : Boolean  := False) return UTF_8_String;
```

Encodes from Wide_Wide_String input, and generates an output encoded in UTF-8 encoding.

```
function Encode (Item       : Wide_Wide_String;
                 Output_BOM : Boolean  := False) return UTF_16_Wide_String;
```

Encodes from Wide_Wide_String input, and generates an output encoded in UTF_16 encoding.

```
function Decode (Item         : UTF_String;
                 Input_Scheme : Encoding_Scheme) return Wide_Wide_String;
```

Decodes from input encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme, and returns the corresponding Wide_Wide_String value.

```
function Decode (Item : UTF_8_String) return Wide_Wide_String;
```

Decodes from input encoded in UTF-8, and returns the corresponding Wide_Wide_String value.

```
function Decode (Item : UTF_16_Wide_String) return Wide_Wide_String;
```

Decodes from input encoded in UTF-16, and returns the corresponding Wide_Wide_String value.

Implementation Advice

If an implementation supports other encoding schemes, similar children of Ada.Strings should be defined.

NOTE

14 A BOM (Byte-Order Mark, code position 16#FEFF#) can be included in a file or other entity to indicate the encoding; it is skipped when decoding. Typically, only the first line of a file or other entity contains a BOM. When decoding, the Encoding function can be called on the first line to determine the encoding; this encoding will then be used in subsequent calls to Decode to convert all of the lines to an internal format.

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.16 The Package Directories

Insert after paragraph 20: [AI05-0049-1]

```
function Compose (Containing_Directory : in String := "";
                  Name                 : in String;
                  Extension            : in String := "") return String;
```

the new paragraphs:

```
type Name_Case_Kind is (Unknown, Case_Sensitive, Case_Insensitive,
                        Case_Preserving);

function Name_Case_Equivalence (Name : in String) return Name_Case_Kind;
```

Insert after paragraph 82: [AI05-0049-1]

Returns the name of the external file with the specified Containing_Directory, Name, and Extension. If Extension is the null string, then Name is interpreted as a simple name; otherwise Name is interpreted as a base name. The exception Name_Error is propagated if the string given as Containing_Directory is not null and does not allow the identification of a directory, or if the string given as Extension is not null and is not a possible extension, or if the string given as Name is not a possible simple name (if Extension is null) or base name (if Extension is non-null).

the new paragraphs:

```
function Name_Case_Equivalence (Name : in String) return Name_Case_Kind;
```

Returns the file name equivalence rule for the directory containing Name. Raises Name_Error if Name is not a full name. Returns Case_Sensitive if file names that differ only in the case of letters are considered different names. If file names that differ only in the case of letters are considered the same name, then Case_Preserving is returned if the name has the case of the file name used when a file is created; and Case_Insensitive otherwise. Returns Unknown if the file name equivalence is not known.

A.16.1 The Package Directories.Hierarchical_File_Names

Insert new clause: [AI05-0049-1]

The library package Directories.Hierarchical_File_Names is an optional package providing operations for file name construction and decomposition for targets with hierarchical file naming.

Static Semantics

If provided, the library package Directories.Hierarchical_File_Names has the following declaration:

```
package Ada.Directories.Hierarchical_File_Names is

   function Is_Simple_Name (Name : in String) return Boolean;

   function Is_Root_Directory_Name (Name : in String) return Boolean;

   function Is_Parent_Directory_Name (Name : in String) return Boolean;

   function Is_Current_Directory_Name (Name : in String) return Boolean;

   function Is_Full_Name (Name : in String) return Boolean;

   function Is_Relative_Name (Name : in String) return Boolean;

   function Simple_Name (Name : in String) return String
      renames Ada.Directories.Simple_Name;

   function Containing_Directory (Name : in String) return String
      renames Ada.Directories.Containing_Directory;

   function Initial_Directory (Name : in String) return String;

   function Relative_Name (Name : in String) return String;

   function Compose (Directory      : in String := "";
                     Relative_Name  : in String;
                     Extension      : in String := "") return String;

end Ada.Directories.Hierarchical_File_Names;
```

In addition to the operations provided in package Directories.Hierarchical_File_Names, operations in package Directories can be used. In particular, functions Full_Name, Base_Name, and Extension are usable with hierarchical file names.

```
function Is_Simple_Name (Name : in String) return Boolean;
```

Returns True if Name is a simple name, and returns False otherwise.

```
function Is_Root_Directory_Name (Name : in String) return Boolean;
```

Returns True if Name is syntactically a root (a directory that cannot be decomposed further), and returns False otherwise.

```
function Is_Parent_Directory_Name (Name : in String) return Boolean;
```

Returns True if Name can be used to indicate symbolically the parent directory of any directory, and returns False otherwise.

```
function Is_Current_Directory_Name (Name : in String) return Boolean;
```

Returns True if Name can be used to indicate symbolically the the directory itself for any directory, and returns False otherwise.

```
function Is_Full_Name (Name : in String) return Boolean;
```

Returns True if the leftmost directory part of Name is a root, and returns False otherwise.

```
function Is_Relative_Name (Name : in String) return Boolean;
```

Returns True if Name allows the identification of an external file but is not a full name, and returns False otherwise.

```
function Initial_Directory (Name : in String) return String;
```

Returns the leftmost directory part in Name. That is a root directory name (for a full name), or one of a parent directory name, a current directory name, or a simple name (for a relative name). The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

```
function Relative_Name (Name : in String) return String;
```

Returns the entire file name except the Initial_Directory portion. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files), or if Name has a single part (this includes if any of Is_Simple_Name, Is_Root_Directory_Name, Is_Parent_Directory_Name, or Is_Current_Directory_Name are True).

```
function Compose (Directory      : in String := "";
                  Relative_Name  : in String;
                  Extension      : in String := "") return String;
```

Returns the name of the external file with the specified Directory, Relative_Name, and Extension. The exception Name_Error is propagated if the string given as Directory is not the null string and does not allow the identification of a directory, or if Is_Relative_Name (Relative_Name) is False, or if the string given as Extension is not the null string and is not a possible extension, or if Extension is not the null string and Simple_Name (Relative_Name) is not a base name.

The result of Compose is a full name if Is_Full_Name (Directory) is True; result is a relative name otherwise.

Implementation Advice

Directories.Hierarchical_File_Names should be provided for systems with hierarchical file naming, and should not be provided on other systems.

NOTES:

42 These operations operate on file names, not external files. The files identified by these operations do not need to exist. Name_Error is raised only as specified or if the file name is malformed and cannot possibly identify a file. The result of these operations depends only on their parameters.

43 Containing_Directory raises Use_Error if Name does not have a containing directory, including when any of Is_Simple_Name, Is_Root_Directory_Name, Is_Parent_Directory_Name, or Is_Current_Directory_Name are True.

A.18 Containers

Insert after paragraph 4: [AI05-0001-1; 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 function is used to provide an ordering for a container, it is generally required to define a strict weak ordering. A function "<" defines a *strict weak ordering* if it is irreflexive, asymmetric, transitive, and in addition, if $x < y$ for any values x and y, then for all other values z, $(x < z)$ or $(z < y)$.

A.18.1 The Package Containers

Insert after paragraph 5: [AI05-0001-1]

```
type Count_Type is range 0 .. implementation-defined;
```

the new paragraph:

```
Capacity_Error : exception;
```

A.18.2 The Generic 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);
```

Insert after paragraph 34: [AI05-0001-1]

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

the new paragraphs:

```
   procedure Assign (Target : in out Vector; Source : in Vector);

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

Insert after paragraph 88: [AI05-0001-1]

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

the new paragraph:

Vector'Write writes the length of Vector elements to the stream. Vector'Read reads the length of Vector elements from the stream.

Insert after paragraph 89: [AI05-0001-1]

No_Index represents a position that does not correspond to any element. The subtype Extended_Index includes the indices covered by Index_Type plus the value No_Index and, if it exists, the successor to the Index_Type'Last.

the new paragraph:

If an operation attempts to modify the vector such that the position of the last element would be greater than Index_Type'Last, then the operation propagates Constraint_Error.

Insert after paragraph 93: [AI05-0001-1]

- it finalizes *V*; or

the new paragraph:

- it calls the Assign procedure with *V* as the Target parameter; or

Replace paragraph 115: [AI05-0001-1]

Reserve_Capacity allocates new internal data structures such that the length of the resulting vector can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then copies the elements into the new data structures and deallocates the old data structures. Any exception raised during allocation is propagated and Container is not modified.

by:

If the capacity of Container is already greater than or equal to Capacity then Reserve_Capacity has no effect. Otherwise, Reserve_Capacity allocates additional storage as necessary to ensure that the length of the resulting vector can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then, as necessary, moves elements into the new storage and deallocates any storage no longer needed. Any exception raised during allocation is propagated and Container is not modified.

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.

Insert after paragraph 147: [AI05-0001-1]

The element designated by Position is not an empty element after successful completion of this operation.

the new paragraphs:

```
procedure Assign (Target : in out Vector; Source : in Vector);
```

If Target denotes the same object as Source, the operation has no effect. If the length of Source is greater than the capacity of Target, Reserve_Capacity is called with the length of Source as the capacity. Each element of Source is assigned to the corresponding element of Target.

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

Returns a vector whose elements are initialized from the corresponding elements of Source. If Capacity is 0, then the vector capacity is the length of Source; if Capacity is equal to or greater than the length of Source, the vector capacity is at least the specified value. Otherwise, the operation propagates Capacity_Error.

Replace paragraph 149: [AI05-0001-1]

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target); then, each element from Source is removed from Source and inserted into Target in the original order. The length of Source is 0 after a successful call to Move.

by:

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Reserve_Capacity (Target, Length (Source)) and then Clear (Target); then, each element from Source is removed from Source and inserted into Target in the original order. The length of Source is 0 after a successful call to Move.

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.

Insert after paragraph 249: [AI05-0160-1]

- The vector that contains the element it designates has been finalized;

the new paragraph:

- The vector that contains the element it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement;

Replace paragraph 251: [AI05-0160-1]

- The element it designates has been deleted.

by:

- The element it designates has been deleted or removed from the vector that contains the element.

A.18.3 The Generic 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);
```

Insert after paragraph 17: [AI05-0001-1]

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

the new paragraphs:

```
   procedure Assign (Target : in out List; Source : in List);

   function Copy (Source : List) return List;
```

Insert after paragraph 60: [AI05-0001-1]

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

the new paragraph:

List'Write writes the length of List elements to the stream. List'Read reads the length of List elements from the stream.

Insert after paragraph 65: [AI05-0001-1]

- it finalizes *L*; or

the new paragraph:

- it calls the Assign procedure with *L* as the Target parameter; or

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.

Insert after paragraph 86: [AI05-0001-1]

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

the new paragraphs:

```
procedure Assign (Target : in out List; Source : in List);
```

If Target denotes the same object as Source, the operation has no effect. Otherwise, it clears Target, and each element of Source is assigned to the corresponding element of Target.

```
function Copy (Source : List) return List;
```

Returns a list whose elements match the elements of Source.

Replace paragraph 88: [AI05-0001-1]

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target). Then, the nodes in Source are moved to Target (in the original order). The length of Target is set to the length of Source, and the length of Source is set to 0.

by:

If Target denotes the same object as Source, then Move has no effect. Otherwise, equivalent to Assign (Target, Source) followed by Clear (Source).

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.

Insert after paragraph 154: [AI05-0160-1]

- The list that contains the element it designates has been finalized;

the new paragraph:

- The list that contains the element it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement;

Replace paragraph 156: [AI05-0160-1]

- The element it designates has been deleted.

by:

- The element it designates has been removed from the list that contains the element.

A.18.4 Maps

Insert after paragraph 10: [AI05-0001-1]

- it finalizes *M*; or

the new paragraph:

- it calls the Assign procedure with *M* as the Target parameter; or

Insert after paragraph 19: [AI05-0001-1]

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

the new paragraph:

Map'Write writes the length of Map elements to the stream. Map'Read reads the length of Map elements from the stream.

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 41: [AI05-0001-1]

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

the new paragraph:

```
procedure Assign (Target : in out Map; Source : in Map);
```

If Target denotes the same object as Source, the operation has no effect. Otherwise, each key/element pair of Source is assigned to the corresponding key/element pair of Target.

Replace paragraph 43: [AI05-0001-1]

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target). Then, each node from Source is removed from Source and inserted into Target. The length of Source is 0 after a successful call to Move.

by:

If Target denotes the same object as Source, then Move has no effect. Otherwise, equivalent to Assign (Target, Source) followed by Clear (Source).

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.

Insert after paragraph 77: [AI05-0160-1]

- The map that contains the node it designates has been finalized;

the new paragraph:

- The map that contains the node it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement;

Replace paragraph 79: [AI05-0160-1]

- The node it designates has been deleted from the map.

by:

- The node it designates has been removed from the map that contains the element.

A.18.5 The Generic 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);
```

Insert after paragraph 17: [AI05-0001-1]

```
   procedure Update_Element
     (Container : in out Map;
      Position  : in     Cursor;
      Process   : not null access procedure
                        (Key     : in     Key_Type;
```

```
                                Element : in out Element_Type));
```

the new paragraphs:

```
   procedure Assign (Target : in out Map; Source : in Map);

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

Insert after paragraph 53: [AI05-0001-1]

In addition to the semantics described in A.18.4, Clear does not affect the capacity of Container.

the new paragraph:

```
procedure Assign (Target : in out Map; Source : in Map);
```

In addition to the semantics described in A.18.4, if the length of Source is greater than the capacity of Target, Reserve_Capacity is called with the length of Source as the capacity before assigning any elements.

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

Returns a map whose keys and elements are initialized from the keys and elements of Source. If Capacity is 0, then the map capacity is the length of Source; if Capacity is equal to or greater than the length of Source, the map capacity is at least the specified value. Otherwise, the operation propagates Capacity_Error.

A.18.6 The Generic 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);
```

Insert after paragraph 16: [AI05-0001-1]

```
   procedure Update_Element
     (Container : in out Map;
      Position  : in     Cursor;
      Process   : not null access procedure
                        (Key     : in     Key_Type;
                         Element : in out Element_Type));
```

the new paragraphs:

```
   procedure Assign (Target : in out Map; Source : in Map);

   function Copy (Source : Map) return Map;
```

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.

Insert after paragraph 58: [AI05-0001-1]

The first node of a nonempty map is the one whose key is less than the key of all the other nodes in the map. The last node of a nonempty map is the one whose key is greater than the key of all the other elements in the map. The successor of a node is the node with the smallest key that is larger than the key of the given node. The predecessor of a node is the node with the largest key that is smaller than the key of the given node. All comparisons are done using the generic formal "<" operator for keys.

the new paragraphs:

```
function Copy (Source : Map) return Map;
```

Returns a map whose keys and elements are initialized from the corresponding keys and elements of Source.

A.18.7 Sets

Insert after paragraph 10: [AI05-0001-1]

- it finalizes *S*; or

the new paragraph:

- it calls the Assign procedure with *S* as the Target parameter; or

Insert after paragraph 18: [AI05-0001-1]

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

the new paragraph:

Set'Write writes the length of Set elements to the stream. Set'Read reads the length of Set elements from the stream.

Replace paragraph 36: [AI05-0001-1; 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.

```
procedure Assign (Target : in out Set; Source : in Set);
```

If Target denotes the same object as Source, the operation has no effect. Otherwise, each element of Source is assigned to the corresponding element of Target.

Replace paragraph 38: [AI05-0001-1]

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first clears Target. Then, each element from Source is removed from Source and inserted into Target. The length of Source is 0 after a successful call to Move.

by:

If Target denotes the same object as Source, then Move has no effect. Otherwise, equivalent to Assign (Target, Source) followed by Clear (Source).

Replace paragraph 58: [AI05-0004-1]

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

by:

Intersection deletes from Target the elements of Target that are not equivalent to 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.

Insert after paragraph 98: [AI05-0160-1]

- The set that contains the element it designates has been finalized;

the new paragraph:

- The set that contains the element it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement;

Replace paragraph 100: [AI05-0160-1]

- The element it designates has been deleted from the set.

by:

- The element it designates has been removed from the set that contains the element.

A.18.8 The Generic 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 17: [AI05-0001-1]

```
   procedure Query_Element
     (Position  : in     Cursor;
      Process   : not null access procedure (Element : in Element_Type));
```

the new paragraphs:

```
   procedure Assign (Target : in out Set; Source : in Set);

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

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.

Insert after paragraph 75: [AI05-0001-1]

In addition to the semantics described in A.18.7, Clear does not affect the capacity of Container.

the new paragraph:

```
procedure Assign (Target : in out Set; Source : in Set);
```

In addition to the semantics described in A.18.7, if the length of Source is greater than the capacity of Target, Reserve_Capacity is called with the length of Source as the capacity before assigning any elements.

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

Returns a set whose elements are initialized from the elements of Source. If Capacity is 0, then the set capacity is the length of Source; if Capacity is equal to or greater than the length of Source, the set capacity is at least the specified value. Otherwise, the operation propagates Capacity_Error.

A.18.9 The Generic 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);
```

Insert after paragraph 16: [AI05-0001-1]

```
   procedure Query_Element
     (Position  : in     Cursor;
      Process   : not null access procedure (Element : in Element_Type));
```

the new paragraphs:

```
   procedure Assign (Target : in out Set; Source : in Set);

   function Copy (Source : Set) return Set;
```

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.

Insert after paragraph 81: [AI05-0001-1]

The first element of a nonempty set is the one which is less than all the other elements in the set. The last element of a nonempty set is the one which is greater than all the other elements in the set. The successor of an element is the smallest element that is larger than the given element. The predecessor of an element is the largest element that is smaller than the given element. All

comparisons are done using the generic formal "<" operator for elements.

the new paragraphs:

```
function Copy (Source : Set) return Set;
```

Returns a set whose elements are initialized from the corresponding elements of Source.

A.18.10 The Generic Package Containers.Multiway_Trees

Insert new clause: [AI05-0136-1]

The language-defined generic package Containers.Multiway_Trees provides private types Tree and Cursor, and a set of operations for each type. A multiway tree container is well-suited to represent nested structures.

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

A *subtree* is a particular node (which *roots the subtree*) and all of its child nodes (including all of the children of the child nodes, recursively). There is a special node, the *root*, which is always present and has no associated element value. The root node provides a place to add nodes to an otherwise empty tree and represents the bottom of the tree.

A node that has no children is called a *leaf node*. The *ancestors* of a node are the parent node, the parent of the parent node, and so on until a node with no parent is reached. Similarly, the *descendants* of a node are the child nodes, the children of each child node, and so on.

The nodes of a subtree could be visited in several different orders. For a *depth-first order*, the last step of visiting a node is to visit the nodes of its child list in order, recursively.

AARM Ramification: For the depth-first order, when each child node is visited, the child list of the child node is visited before the next sibling of the child node is visited.

Static Semantics

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

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

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

   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);

   Empty_Tree : constant Tree;

   No_Element : constant Cursor;

   function Equal_Subtree (Left_Position : Cursor;
                           Right_Position: Cursor) return Boolean;

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

   function Is_Empty (Container : Tree) return Boolean;

   function Node_Count (Container : Tree) return Count_Type;

   function Subtree_Node_Count (Position : Cursor) return Count_Type;

   function Depth (Position : Cursor) return Count_Type;

   function Is_Root (Position : Cursor) return Boolean;

   function Is_Leaf (Position : Cursor) return Boolean;

   function Root (Container : Tree) return Cursor;
```

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

      function Element (Position : Cursor) return Element_Type;

      procedure Replace_Element (Container : in out Tree;
                                 Position  : in     Cursor;
                                 New_Item  : in     Element_Type);

      procedure Query_Element
        (Position : in Cursor;
         Process  : not null access procedure (Element : in Element_Type));

      procedure Update_Element
        (Container : in out Tree;
         Position  : in     Cursor;
         Process   : not null access procedure
                        (Element : in out Element_Type));

      procedure Assign (Target : in out Tree; Source : in Tree);

      function Copy (Source : Tree) return Tree;

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

      procedure Delete_Leaf (Container : in out Tree;
                             Position  : in out Cursor);

      procedure Delete_Subtree (Container : in out Tree;
                                Position  : in out Cursor);

      procedure Swap (Container : in out Tree;
                      I, J      : in     Cursor);

      function Find (Container : Tree;
                     Item      : Element_Type)
         return Cursor;

      function Find_In_Subtree (Container : Tree;
                                Item      : Element_Type;
                                Position  : Cursor)
         return Cursor;

      function Ancestor_Find (Container : Tree;
                              Item      : Element_Type;
                              Position  : Cursor)
         return Cursor;

      function Contains (Container : Tree;
                         Item      : Element_Type) return Boolean;

      function Has_Element (Position : Cursor) return Boolean;

      procedure Iterate
        (Container : in Tree;
         Process   : not null access procedure (Position : in Cursor));

      procedure Iterate_Subtree
        (Position  : in Cursor;
         Process   : not null access procedure (Position : in Cursor));

      function Child_Count (Parent : Cursor) return Count_Type;

      function Child_Depth (Parent, Child : Cursor) return Count_Type;

      procedure Insert_Child (Container : in out Tree;
                              Parent    : in     Cursor;
                              Before    : in     Cursor;
                              New_Item  : in     Element_Type;
                              Count     : in     Count_Type := 1);

      procedure Insert_Child (Container : in out Tree;
                              Parent    : in     Cursor;
```

```
                        Before    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1);

procedure Insert_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1);

procedure Prepend_Child (Container : in out Tree;
                         Parent    : in     Cursor;
                         New_Item  : in     Element_Type;
                         Count     : in     Count_Type := 1);

procedure Append_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1);

procedure Delete_Children (Container : in out Tree;
                           Parent    : in     Cursor);

procedure Copy_Subtree (Target   : in out Tree;
                        Parent   : in     Cursor;
                        Before   : in     Cursor;
                        Source   : in     Cursor);

procedure Splice_Subtree (Target   : in out Tree;
                          Parent   : in     Cursor;
                          Before   : in     Cursor;
                          Source   : in out Tree;
                          Position : in out Cursor);

procedure Splice_Subtree (Container: in out Tree;
                          Parent   : in     Cursor;
                          Before   : in     Cursor;
                          Position : in     Cursor);

procedure Splice_Children (Target          : in out Tree;
                           Target_Parent   : in     Cursor;
                           Before          : in     Cursor;
                           Source          : in out Tree;
                           Source_Parent   : in     Cursor);

procedure Splice_Children (Container       : in out Tree;
                           Target_Parent   : in     Cursor;
                           Before          : in     Cursor;
                           Source_Parent   : in     Cursor);

function Parent (Position : Cursor) return Cursor;

function First_Child (Parent : Cursor) return Cursor;

function First_Child_Element (Parent : Cursor) return Element_Type;

function Last_Child (Parent : Cursor) return Cursor;

function Last_Child_Element (Parent : Cursor) return Element_Type;

function Next_Sibling (Position : Cursor) return Cursor;

function Previous_Sibling (Position : Cursor) return Cursor;

procedure Next_Sibling (Position : in out Cursor);

procedure Previous_Sibling (Position : in out Cursor);

procedure Iterate_Children
  (Container : in Tree;
   Parent    : in     Cursor;
   Process   : not null access procedure (Position : in Cursor));
```

```
      procedure Reverse_Iterate_Children
        (Container : in Tree;
         Parent    : in     Cursor;
         Process   : not null access procedure (Position : in Cursor));

   private
      ... -- not specified by the language
   end Ada.Containers.Multiway_Trees;
```

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 functions Find, Reverse_Find, Equal_Subtree, and "=" on tree values return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions Find, Reverse_Find, Equal_Subtree, and "=" on tree values are unspecified.

The type Tree is used to represent trees. The type Tree needs finalization (see 7.6).

Empty_Tree represents the empty Tree object. It contains only the root node (Count (Empty_Tree) returns 1). If an object of type Tree is not otherwise initialized, it is initialized to the same value as Empty_Tree.

No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

The predefined "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

Tree'Write writes Tree.Count-1 elements to the stream. Tree'Read reads Tree.Count-1 elements from the stream.

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 cursors" of a container because they depend on the set of elements of the container remaining constant, and others 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 cursors* of a tree object *T* if:

- it inserts or deletes elements of *T*, that is, it calls the Clear, Delete_Leaf, Insert_Child, or Delete_Children procedures with *T* as a parameter; or
- it reorders the elements of *T*, that is, it calls the Splice_Subtree or Splice_Children procedures; or
- it finalizes *T*; or
- it calls Assign with *T* as the Target parameter; or
- it calls the Move procedure with *T* as a parameter.

A subprogram is said to *tamper with elements* of a tree object *T* if:

- it tampers with cursors of *T*; or
- it replaces one or more elements of *T*, that is, it calls the Replace_Element or Swap procedures with *T* as a parameter.

```
function Equal_Subtree (Left_Position : Cursor;
                        Right_Position: Cursor) return Boolean;
```

If Left_Position or Right_Position equals No_Element, propagates Constraint_Error. If the number of child nodes of the element designated by Left_Position is different than the number of child nodes of the element designated by Right_Position, the function returns False. If Left_Position designates a root node and Right_Position does not, the function returns False. If Right_Position designates a root node and Left_Position does not, the function returns False. Unless both cursors designate a root node, the elements are compared using the generic formal equality operator. If the result is False, the function returns False. Otherwise, it calls Equal_Subtree on a cursor designating each child element of the element designated by Left_Position and a cursor designated the corresponding child element of the element designated by Right_Position. If any such call returns False, the function returns False; otherwise it returns True. Any exception raised during the evaluation of element equality is propagated.

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

If Left and Right denote the same tree object, then the function returns True. Otherwise, it calls Equal_Subtree with a cursor designating the root node of Left and Right; the result is returned. Any exception raised during the evaluation of Equal_Subtree is propagated.

```
function Node_Count (Container : Tree) return Count_Type;
```

Node_Count returns the number of nodes in Container.

```
function Subtree_Node_Count (Position : Cursor) return Count_Type;
```

If Position is No_Element, Subtree_Count returns 0; otherwise, Subtree_Node_Count returns the number of nodes in the subtree that is rooted by Position.

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

Equivalent to Count (Container) = 1.

```
function Depth (Position : Cursor) return Count_Type;
```

If Position equals No_Element, Depth returns 0; otherwise Depth returns the number of ancestor nodes of the node designated by Position (including the node itself).

```
function Is_Root (Position : Cursor) return Boolean;
```

Is_Root returns True if the Position designates the root node of the tree; and returns False otherwise.

```
function Is_Leaf (Position : Cursor) return Boolean;
```

Is_Leaf returns True if Position designates a node that does not have any child nodes; and returns False otherwise.

```
function Root (Container : Tree) return Cursor;
```

Root returns a cursor that designates the root node of Container.

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

Removes all the elements from Container.

```
function Element (Position : Cursor) return Element_Type;
```

If Position equals No_Element, then Constraint_Error is propagated; if Position designates the root node of a tree, then Program_Error is propagated. Otherwise, Element returns the element designated by Position.

```
procedure Replace_Element (Container : in out Tree;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container (including if it designates the root node), then Program_Error is propagated. Otherwise Replace_Element assigns the value New_Item to the element designated by Position.

```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));
```

If Position equals No_Element, then Constraint_Error is propagated; if Position designates the root node of a tree, then Program_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.

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

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container (including if it designates the root node), then Program_Error is propagated. Otherwise Update_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.

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

```
procedure Assign (Target : in out Tree; Source : in Tree);
```

If Target denotes the same object as Source, the operation has no effect. Otherwise, it calls Clear (Target), and then each element of Source is assigned to a corresponding element in Target.

```
function Copy (Source : Tree) return Tree;
```

Returns a tree with the same structure as Source and whose elements are initialized from the corresponding elements of Source.

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

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target). Then,

the nodes other than the root node in Source are moved to Target (in the same positions). After Move completes, Count(Target) is the number of nodes originally in Source, and Count(Source) is 1.

```
procedure Delete_Leaf (Container : in out Tree;
                       Position  : in out Cursor);
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container (including if it designates the root node), then Program_Error is propagated. If the element designated by position has any child elements, then Constraint_Error is propagated. Otherwise Delete_Leaf removes (from Container) the element designated by Position. Finally, Position is set to No_Element.

```
procedure Delete_Subtree (Container : in out Tree;
                          Position  : in out Cursor);
```

If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container (including if it designates the root node), then Program_Error is propagated. Otherwise Delete removes (from Container) the subtree designated by Position (that is, the node designated by Position and all of the descendant nodes of that node), and Position is set to No_Element.

```
procedure Swap (Container : in out Tree;
                I, J      : in     Cursor);
```

If either I or J equals No_Element, then Constraint_Error is propagated. If either I or J do not designate an element in Container (including if either designates the root node), then Program_Error is propagated. Otherwise, Swap exchanges the values of the elements designated by I and J.

```
function Find (Container : Tree;
               Item      : Element_Type)
   return Cursor;
```

Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the root node. The search checks the tree in a depth-first order. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

```
function Find_In_Subtree (Container : Tree;
                          Item      : Element_Type;
                          Position  : Cursor)
      return Cursor;
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate a node in Container, then Program_Error is propagated. Find_In_Subtree searches a subtree of the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position. The search checks the subtree rooted by Position in a depth-first order. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

```
function Ancestor_Find (Container : Tree;
                        Item      : Element_Type;
                        Position  : Cursor)
   return Cursor;
```

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container (including if it designates the root node), then Program_Error is propagated. Otherwise, Ancestor_Find searches for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position, and checks each ancestor proceeding toward the root of the subtree. If no equal element is found, then Ancestor_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

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

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

```
function Has_Element (Position : Cursor) return Boolean;
```

Returns True if Position designates an element, and returns False otherwise. In particular, Has_Element returns False if the cursor designates the root node or equals No_Element.

```
procedure Iterate
  (Container : in Tree;
   Process   : not null access procedure (Position : in Cursor));
```

Iterate calls Process.**all** with a cursor that designates each element in Container, starting with the root node and proceeding in a depth-first order. Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process.**all** is propagated.

```
procedure Iterate_Subtree
  (Position  : in Cursor;
```

```
   Process   : not null access procedure (Position : in Cursor));
```

If Position equals No_Element, then Constraint_Error is propagated. Iterate calls Process.**all** with a cursor that designates each element in the subtree rooted by the node designated by Position, starting with the node designated by Position and proceeding in a depth-first order. Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process.**all** is propagated.

```
function Child_Count (Parent : Cursor) return Count_Type;
```

Child_Count returns the number of child nodes of the node designated by Parent.

```
function Child_Depth (Parent, Child : Cursor) return Count_Type;
```

If Child or Parent is equal to No_Element, then Constraint_Error is propagated. Otherwise, Child_Depth returns the number of ancestor nodes of Child (including Child itself), up to but not including Parent; in this case Program_Error is propagated if Parent is not an ancestor of Child.

```
procedure Insert_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1);
```

If Parent equals No_Element, then Constraint_Error is propagated. If Parent does not designate a node in Container, then Program_Error is propagated. If Before is not equal to No_Element, and does not designate a node in Container, then Program_Error is propagated. If Before is not equal to No_Element, and Container.Parent (Before) is not equal to Parent, then Constraint_Error is propagated. Otherwise, Insert_Child allocates Count nodes containing copies of New_Item and inserts them as children of Parent. If Parent already has child nodes then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

```
procedure Insert_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1);
```

If Parent equals No_Element, then Constraint_Error is propagated. If Parent does not designate a node in Container, then Program_Error is propagated. If Before is not equal to No_Element, and does not designate a node in Container, then Program_Error is propagated. If Before is not equal to No_Element, and Container.Parent (Before) is not equal to Parent, then Constraint_Error is propagated. Otherwise, Insert_Child allocates Count nodes containing copies of New_Item and inserts them as children of Parent. If Parent already has child nodes then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. Position designates the first newly-inserted node. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

```
procedure Insert_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1);
```

If Parent equals No_Element, then Constraint_Error is propagated. If Parent does not designate a node in Container, then Program_Error is propagated. If Before is not equal to No_Element, and does not designate a node in Container, then Program_Error is propagated. If Before is not equal to No_Element, and Container.Parent (Before) is not equal to Parent, then Constraint_Error is propagated. Otherwise, Insert_Child allocates Count nodes, the elements contained in the new nodes are initialized by default (see 3.3.1), and the new nodes are inserted as children of Parent. If Parent already has child nodes then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. Position designates the first newly-inserted node. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

```
procedure Prepend_Child (Container : in out Tree;
                         Parent    : in     Cursor;
                         New_Item  : in     Element_Type;
                         Count     : in     Count_Type := 1);
```

Equivalent to Insert_Child (Container, Parent, First_Child (Container, Parent), New_Item, Count).

```
procedure Append_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1);
```

Equivalent to Insert_Child (Container, Parent, Last_Child (Container, Parent), New_Item, Count).

```
procedure Delete_Children (Container : in out Tree;
                           Parent    : in     Cursor);
```

If Parent equals No_Element, then Constraint_Error is propagated. If Parent does not designate a node in Container, Program_Error is propagated. Otherwise, Delete_Children removes (from Container) all of the child elements of Parent along with their dependent elements.

```
procedure Copy_Subtree (Target   : in out Tree;
                        Parent   : in     Cursor;
                        Before   : in     Cursor;
                        Source   : in     Cursor);
```

If Parent equals No_Element, then Constraint_Error is propagated. If Parent does not designate a node in Target, then Program_Error is propagated. If Before is not equal to No_Element, and does not designate a node in Target, then Program_Error is propagated. If Before is not equal to No_Element, and Target.Parent (Before) is not equal to Parent, then Constraint_Error is propagated. If Source designates a root node, then Constraint_Error is propagated. If Source is equal to No_Element, then the operation has no effect. Otherwise, the subtree rooted by Source (which can be from any tree; it does not have to be a subtree of Target) is copied (new nodes are allocated to create a new subtree with the same structure as the Source subtree, with each element initialized from the corresponding element of the Source subtree) and inserted into Target as a child of Parent. If Parent already has child nodes then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. The parent of the newly created subtree is set to Parent, and the overall count of Target is incremented by Subtree_Count (Source). Any exception raised during allocation of internal storage is propagated, and Container is not modified.

```
procedure Splice_Subtree (Target   : in out Tree;
                          Parent   : in     Cursor;
                          Before   : in     Cursor;
                          Source   : in out Tree;
                          Position : in out Cursor);
```

If Parent equals No_Element, then Constraint_Error is propagated. If Parent does not designate a node in Target, then Program_Error is propagated. If Before is not equal to No_Element, and does not designate a node in Target, then Program_Error is propagated. If Before is not equal to No_Element, and Target.Parent (Before) is not equal to Parent, then Constraint_Error is propagated. If Position equals No_Element, Constraint_Error is propagated. If Position does not designate a node in Source or designates a root node, then Program_Error is propagated. If Source denotes the same object as Target, then if Position designates an ancestor of Parent or is equal to Parent, Constraint_Error is propagated; else the subtree rooted by the element designated by Position is moved to be a child of Parent. If Parent already has child nodes then the moved nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved nodes are inserted after the last existing child node of Parent. In each of these cases, Position and the count of Target are unchanged, and the parent of the element designated by Position is set to Parent.

Otherwise (if Source does not denote the same object as Target), the subtree designated by Position is removed from Source and moved to Target. The subtree is inserted as a child of Parent. If Parent already has child nodes then the moved nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved nodes are inserted after the last existing child node of Parent. In each of these cases, the count of Target is incremented by Subtree_Count (Position), and the count of Source is decremented by Subtree_Count (Position), Position is updated to represent an element in Target.

```
procedure Splice_Subtree (Container: in out Tree;
                          Parent   : in     Cursor;
                          Before   : in     Cursor;
                          Position : in     Cursor);
```

If Parent equals No_Element, then Constraint_Error is propagated. If Parent does not designate a node in Container, then Program_Error is propagated. If Before is not equal to No_Element, and does not designate a node in Container, then Program_Error is propagated. If Before is not equal to No_Element, and Target.Parent (Before) is not equal to Parent, then Constraint_Error is propagated. If Position equals No_Element or designates a root node, Constraint_Error is propagated. If Position does not designate a node in Container, then Program_Error is propagated. If Position equals Before, there is no effect. If Position designates an ancestor of Parent or is equal to Parent, Constraint_Error is propagated. Otherwise, the subtree rooted by the element designated by Position is moved to be a child of Parent. If Parent already has child nodes then the moved nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved nodes are inserted after the last existing child node of Parent. The parent of the element designated by Position is set to Parent.

```
procedure Splice_Children (Target          : in out Tree;
                           Target_Parent   : in     Cursor;
                           Before          : in     Cursor;
                           Source          : in out Tree;
```

```
                                Source_Parent   : in     Cursor);
```

If Target_Parent equals No_Element, then Constraint_Error is propagated. If Target_Parent does not designate a node in Target, then Program_Error is propagated. If Before is not equal to No_Element, and does not designate an element in Target, then Program_Error is propagated. If Source_Parent equals No_Element, then Constraint_Error is propagated. If Source_Parent does not designate a node in Source, then Program_Error is propagated. If Before is not equal to No_Element, and Target.Parent (Before) is not equal to Target_Parent, then Constraint_Error is propagated.

If Source denotes the same object as Target, then:

- if Target_Parent equals Source_Parent there is no effect; else
- if Source_Parent is an ancestor of Target_Parent, then Constraint_Error is propagated; else
- the child elements (and their descendants) of Source_Parent are moved to be child elements of Target_Parent. If Target_Parent already has child elements then the moved elements are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved elements are inserted after the last existing child node of Target_Parent. The parent of each moved child element is set to Target_Parent.

Otherwise (if Source does not denote the same object as Target), the child elements (and their descendants) of Source_Parent are removed from Source and moved to Target. The child elements are inserted as children of Target_Parent. If Target_Parent already has child elements then the moved elements are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved elements are inserted after the last existing child node of Target_Parent. In each of these cases, the overall count of Target is incremented by Subtree_Count (Source_Parent)-1, and overall count of Source is decremented by Subtree_Count (Source_Parent)-1.

```
procedure Splice_Children (Container       : in out Tree;
                           Target_Parent   : in     Cursor;
                           Before          : in     Cursor;
                           Source_Parent   : in     Cursor);
```

If Target_Parent equals No_Element, then Constraint_Error is propagated. If Target_Parent does not designate a node in Container, then Program_Error is propagated. If Before is not equal to No_Element, and does not designate an element in Container, then Program_Error is propagated. If Source_Parent equals No_Element, then Constraint_Error is propagated. If Source_Parent does not designate a node in Container, then Program_Error is propagated. If Before is not equal to No_Element, and Container.Parent (Before) is not equal to Target_Parent, then Constraint_Error is propagated. If Target_Parent equals Source_Parent there is no effect. If Source_Parent is an ancestor of Target_Parent, then Constraint_Error is propagated. Otherwise the child elements (and their descendants) of Source_Parent are moved to be child elements of Target_Parent. If Target_Parent already has child elements then the moved elements are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved elements are inserted after the last existing child node of Target_Parent. The parent of each moved child element is set to Target_Parent.

```
function Parent (Position : Cursor) return Cursor;
```

If Position is equal to No_Element or designates a root node, No_Element is returned. Otherwise, a cursor designating the parent node of the node designated by Position is returned.

```
function First_Child (Parent : Cursor) return Cursor;
```

If Parent is equal to No_Element, then Constraint_Error is propagated. Otherwise, First_Child returns a cursor designating the first child node of the node designated by Parent; if there is no such node, No_Element is returned.

```
function First_Child_Element (Parent : Cursor) return Element_Type;
```

Equivalent to Element (First_Child (Parent)).

```
function Last_Child (Parent : Cursor) return Cursor;
```

If Parent is equal to No_Element, then Constraint_Error is propagated. Otherwise, Last_Child returns a cursor designating the last child node of the node designated by Parent; if there is no such node, No_Element is returned.

```
function Last_Child_Element (Parent : Cursor) return Element_Type;
```

Equivalent to Element (Last_Child (Parent)).

```
function Next_Sibling (Position : Cursor) return Cursor;
```

If Position equals No_Element or designates the last child node of its parent, then Next_Sibling returns the value No_Element. Otherwise, it returns a cursor that designates the successor (with the same parent) of the node designated by Position.

```
function Previous_Sibling (Position : Cursor) return Cursor;
```

If Position equals No_Element or designates the first child node of its parent, then Previous_Sibling returns the value No_Element. Otherwise, it returns a cursor that designates the predecessor (with the same parent) of the node designated by Position.

```
procedure Next_Sibling (Position : in out Cursor);
```

Equivalent to Position := Next_Sibling (Position);

```
procedure Previous_Sibling (Position : in out Cursor);
```

Equivalent to Position := Previous_Sibling (Position);

```
procedure Iterate_Children
  (Container : in Tree;
   Parent    : in Cursor;
   Process   : not null access rocedure (Position : in Cursor));
```

If Parent equals No_Element, then Constraint_Error is propagated. If Parent does not designate a node in Container, then Program_Error is propagated.

Iterate_Children calls Process.**all** with a cursor that designates each child node of Parent, starting with the first child node and moving the cursor as per the Next_Sibling function.

Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process.**all** is propagated.

```
procedure Reverse_Iterate_Children
  (Container : in Tree;
   Parent    : in Cursor;
   Process   : not null access procedure (Position : in Cursor));
```

If Parent equals No_Element, then Constraint_Error is propagated. If Parent does not designate a node in Container, then Program_Error is propagated.

Reverse_Iterate_Children calls Process.**all** with a cursor that designates each child node of Parent, starting with the last child node and moving the cursor as per the Previous_Sibling function.

Program_Error is propagated if Process.**all** tampers with the cursors of Container. Any exception raised by Process.**all** is propagated.

Bounded (Run-Time) 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 Tree parameter to the operation. Either Program_Error is raised, or the operation works as defined on the value of the Tree either prior to, or subsequent to, some or all of the modifications to the Tree.

It is a bounded error to call any subprogram declared in the visible part of Containers.Multiway_Trees 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.

Erroneous Execution

A Cursor value is invalid if any of the following have occurred since it was created:

- The tree that contains the element it designates has been finalized;
- The tree that contains the element it designates has been used as the Source or Target of a call to Move;
- The tree that contains the element it designates has been used as the Target of a call to Assign or the target of an assignment_statement;
- The element it designates has been removed from the tree that previously contained the element.

The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Multiway_Trees is called with an invalid cursor parameter.

Implementation Requirements

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

The execution of an assignment_statement for a tree shall have the effect of copying the elements from the source tree object to the target tree object.

Implementation Advice

Containers.Multiway_Trees should be implemented similarly to a multiway tree. In particular, if *N* is the overall number of nodes for a particular tree, then the worst-case time complexity of Element, Parent, First_Child, Last_Child, Insert_Child with Count=1, and Delete should be $O(\log N)$.

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

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

A.18.17 The Generic Package Containers.Indefinite_Multiway_Trees

Insert new clause: [AI05-0136-1]

The language-defined generic package Containers.Indefinite_Multiway_Trees provides a multiway tree with the same operations as the package Containers.Multiway_Trees (see A.18.10), with the difference that the generic formal Element_Type is indefinite.

Static Semantics

The declaration of the generic library package Containers.Indefinite_Multiway_Trees has the same contents and semantics as Containers.Multiway_Trees except:

- The generic formal Element_Type is indefinite.
- The procedure with the profile:

```
procedure Insert_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1);
```

  is omitted.

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

A.18.18 The Generic Package Containers.Indefinite_Holders

Insert new clause: [AI05-0001-1; AI05-0069-1; AI05-0084-1]

The language-defined generic package Containers.Indefinite_Holders provides a 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 Assign (Target : in out Holder; Source : in Holder);

   function Copy (Source : Holder) return Holder;

   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, Update_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 Assign (Target : in out Holder; Source : in Holder);
```

If Target denotes the same object as Source, the operation has no effect. If Source is empty, Clear (Target) is called. Otherwise, Replace_Element (Target, Element (Source)) is called.

```
function Copy (Source : Holder) return Holder;
```

If Source is empty, returns an empty holder; otherwise, returns To_Holder (Element (Source)).

```
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.

A.18.19 The Generic Package Containers.Bounded_Vectors

Insert new clause: [AI05-0001-1; AI05-0160-1; AI05-0184-1]

The language-defined generic package Containers.Bounded_Vectors provides a private type Vector and a set of operations. It provides the same operations as the package Containers.Vectors (see A.18.2), with the difference that the maximum storage is bounded.

Static Semantics

The declaration of the generic library package Containers.Bounded_Vectors has the same contents and semantics as Containers.Vectors except:

- The pragma Preelaborate is replaced with pragma Pure.
- The type Vector is declared with a discriminant that specifies the capacity:

  ```
  type Vector (Capacity : Count_Type) is tagged private;
  ```

- The type Vector needs finalization if and only if type Element_Type needs finalization.
- In function Copy, if the Capacity parameter is equal to or greater than the length of Source, the vector capacity exactly equals the value of the Capacity parameter.
- The description of Reserve_Capacity is replaced with:

  If the specified Capacity is larger than the capacity of Container, then Reserve_Capacity propagates Capacity_Error. Otherwise, the operation has no effect.

Bounded Errors

It is a bounded error to use a bounded vector if it was the target of an assignment_statement whose source was in the middle of an operation that disallows tampering with elements or cursors. Either Program_Error is raised, or the operation proceeds as defined.

Implementation Requirements

For each instance of Containers.Vectors and each instance of Containers.Bounded_Vectors, if the two instances meet the following conditions then the output generated by the Vector'Output or Vector'Write subprograms of either instance shall be readable by the Vector'Input or Vector'Read of the other instance, respectively:

- the Element_Type parameters of the two instances are statically matching subtypes of the same type (note that distinct elaborations of a type declaration result in distinct types); and
- the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of of the two actual Element_Type parameters); and
- the preceding two conditions also hold for the Index_Type parameters of the instances.

Implementation Advice

Bounded vector objects should be implemented without implicit pointers or dynamic allocation.

The implementation advice for procedure Move to minimize copying does not apply.

A.18.20 The Generic Package Containers.Bounded_Doubly_Linked_Lists

Insert new clause: [AI05-0001-1; AI05-0160-1; AI05-0184-1]

The language-defined generic package Containers.Bounded_Doubly_Linked_Lists provides a private type List and a set of operations. It provides the same operations as the package Containers.Doubly_Linked_Lists (see A.18.3), with the difference that the maximum storage is bounded.

Static Semantics

The declaration of the generic library package Containers.Bounded_Doubly_Linked_Lists has the same contents and semantics as Containers.Doubly_Linked_Lists except:

- The pragma Preelaborate is replaced with pragma Pure.
- The type List is declared with a discriminant that specifies the capacity (maximum number of elements) as follows:

```
type List (Capacity : Count_Type) is tagged private;
```

- The type List needs finalization if and only if type Element_Type needs finalization.
- The allocation of internal storage includes a check that the capacity is not exceeded, and Capacity_Error is raised if this check fails.
- In procedure Assign, if Source length is greater than Target capacity, then Capacity_Error is propagated.
- The function Copy is replaced with:

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

  If Capacity is 0, then the list capacity is the length of Source; if Capacity is equal to or greater than the length of Source, the list capacity equals the value of the Capacity parameter; otherwise, the operation propagates Capacity_Error.

- In the three-parameter procedure Splice whose Source has type List, if the sum of the length of Target and the length of Source is greater than the capacity of Target, then Splice propagates Capacity_Error.
- In the four-parameter procedure Splice, if the length of Target equals the capacity of Target, then Splice propagates Capacity_Error.

Bounded Errors

It is a bounded error to use a bounded list if it was the target of an assignment_statement whose source was in the middle of an operation that disallows tampering with elements or cursors. Either Program_Error is raised, or the operation proceeds as defined.

Implementation Requirements

For each instance of Containers.Doubly_Linked_Lists and each instance of Containers.Bounded_Doubly_Linked_Lists, if the two instances meet the following conditions then the output generated by the List'Output or List'Write subprograms of either instance shall be readable by the List'Input or List'Read of the other instance, respectively:

- the Element_Type parameters of the two instances are statically matching subtypes of the same type (note that distinct elaborations of a type declaration result in distinct types); and
- the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of of the two actual Element_Type parameters).

Implementation Advice

Bounded list objects should be implemented without implicit pointers or dynamic allocation.

The implementation advice for procedure Move to minimize copying does not apply.

A.18.21 The Generic Package Containers.Bounded_Hashed_Maps

Insert new clause: [AI05-0001-1; AI05-0160-1; AI05-0184-1]

The language-defined generic package Containers.Bounded_Hashed_Maps provides a private type Map and a set of operations. It provides the same operations as the package Containers.Hashed_Maps (see A.18.5), with the difference that the maximum storage is bounded.

Static Semantics

The declaration of the generic library package Containers.Bounded_Hashed_Maps has the same contents and semantics as Containers.Hashed_Maps except:

- The pragma Preelaborate is replaced with pragma Pure.
- The type Map is declared with discriminants that specify both the capacity (number of elements) and modulus (number of distinct hash values) of the hash table as follows:

```
type Map (Capacity : Count_Type;
          Modulus  : Hash_Type) is tagged private;
```

- The type Map needs finalization if and only if type Key_Type or type Element_Type needs finalization.
- The description of Reserve_Capacity is replaced with:

  If the specified Capacity is larger than the capacity of Container, then Reserve_Capacity propagates Capacity_Error. Otherwise, the operation has no effect.

- An additional operation is added immediately following Reserve_Capacity:

```
function Default_Modulus (Capacity : Count_Type) return Hash_Type;
```

  Default_Modulus returns an implementation-defined value for the number of distinct hash values to be used for the given capacity (maximum number of elements).

- The function Copy is replaced with:

```
function Copy (Source   : Map;
               Capacity : Count_Type := 0;
               Modulus  : Hash_Type := 0) return Map;
```

  Returns a map with key/element pairs initialized from the values in Source. If Capacity is 0, then the map capacity is the length of Source; if Capacity is equal to or greater than the length of Source, the map capacity is the value of the Capacity parameter; otherwise, the operation propagates Capacity_Error. If the Modulus argument is 0, then the map modulus is the value returned by a call to Default_Modulus with the map capacity as its argument; otherwise the map modulus is the value of the Modulus parameter.

Bounded Errors

It is a bounded error to use a bounded map if it was the target of an assignment_statement whose source was in the middle of an operation that disallows tampering with elements or cursors. Either Program_Error is raised, or the operation proceeds as defined.

Implementation Requirements

For each instance of Containers.Hashed_Maps and each instance of Containers.Bounded_Hashed_Maps, if the two instances meet the following conditions then the output generated by the Map'Output or Map'Write subprograms of either instance shall be readable by the Map'Input or Map'Read of the other instance, respectively:

- the Element_Type parameters of the two instances are statically matching subtypes of the same type (note that distinct elaborations of a type declaration result in distinct types); and
- the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of of the two actual Element_Type parameters); and
- the preceding two conditions also hold for the Key_Type parameters of the instances.

Implementation Advice

Bounded map objects should be implemented without implicit pointers or dynamic allocation.

The implementation advice for procedure Move to minimize copying does not apply.

A.18.22 The Generic Package Containers.Bounded_Ordered_Maps

Insert new clause: [AI05-0001-1; AI05-0160-1; AI05-0184-1]

The language-defined generic package Containers.Bounded_Ordered_Maps provides a private type Map and a set of operations. It provides the same operations as the package Containers.Ordered_Maps (see A.18.6), with the difference that the maximum storage is bounded.

Static Semantics

The declaration of the generic library package Containers.Bounded_Ordered_Maps has the same contents and semantics as Containers.Ordered_Maps except:

- The pragma Preelaborate is replaced with pragma Pure.
- The type Map is declared with a discriminant that specifies the capacity (maximum number of elements) as follows:

```
type Map (Capacity : Count_Type) is tagged private;
```

- The type Map needs finalization if and only if type Key_Type or type Element_Type needs finalization.
- The allocation of a new node includes a check that the capacity is not exceeded, and Capacity_Error is raised if this check fails.
- In procedure Assign, if Source length is greater than Target capacity, then Capacity_Error is propagated.
- The function Copy is replaced with:

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

  Returns a map with key/element pairs initialized from the values in Source. If Capacity is 0, then the map capacity is the length of Source; if Capacity is equal to or greater than the length of Source, the map capacity is the specified value; otherwise, the operation propagates Capacity_Error.

Bounded Errors

It is a bounded error to use a bounded map if it was the target of an assignment_statement whose source was in the middle of an operation that disallows tampering with elements or cursors. Either Program_Error is raised, or the operation proceeds as defined.

Implementation Requirements

For each instance of Containers.Ordered_Maps and each instance of Containers.Bounded_Ordered_Maps, if the two instances meet the following conditions then the output generated by the Map'Output or Map'Write subprograms of either instance shall be readable by the Map'Input or Map'Read of the other instance, respectively:

- the Element_Type parameters of the two instances are statically matching subtypes of the same type (note that distinct elaborations of a type declaration result in distinct types); and
- the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of of the two actual Element_Type parameters); and
- the preceding two conditions also hold for the Key_Type parameters of the instances.

Implementation Advice

Bounded map objects should be implemented without implicit pointers or dynamic allocation.

The implementation advice for procedure Move to minimize copying does not apply.

A.18.23 The Generic Package Containers.Bounded_Hashed_Sets

Insert new clause: [AI05-0001-1; AI05-0160-1; AI05-0184-1]

The language-defined generic package Containers.Bounded_Hashed_Sets provides a private type Set and a set of operations. It provides the same operations as the package Containers.Hashed_Sets (see A.18.8), with the difference that the maximum storage is bounded.

Static Semantics

The declaration of the generic library package Containers.Bounded_Hashed_Sets has the same contents and semantics as Containers.Hashed_Sets except:

- The pragma Preelaborate is replaced with pragma Pure.
- The type Set is declared with discriminants that specify both the capacity (number of elements) and modulus (number of distinct hash values) of the hash table as follows:

```
type Set (Capacity : Count_Type;
          Modulus  : Hash_Type) is tagged private;
```

- The type Set needs finalization if and only if type Key_Type or type Element_Type needs finalization.
- The description of Reserve_Capacity is replaced with:

  If the specified Capacity is larger than the capacity of Container, then Reserve_Capacity propagates Capacity_Error. Otherwise, the operation has no effect.

- An additional operation is added immediately following Reserve_Capacity:

```
function Default_Modulus (Capacity : Count_Type) return Hash_Type;
```

  Default_Modulus returns an implementation-defined value for the number of distinct hash values to be used for the given capacity (maximum number of elements).

- The function Copy is replaced with:

```
function Copy (Source   : Set;
               Capacity : Count_Type := 0;
               Modulus  : Hash_Type := 0) return Set;
```

  Returns a set whose elements are initialized from the values in Source. If Capacity is 0, then the set capacity is the length of Source; if Capacity is equal to or greater than the length of Source, the set capacity is the value of the Capacity parameter; otherwise, the operation propagates Capacity_Error. If the Modulus argument is 0, then the set modulus is the value returned by a call to Default_Modulus with the set capacity as its argument; otherwise the set modulus is the value of the Modulus parameter.

Bounded Errors

It is a bounded error to use a bounded set if it was the target of an assignment_statement whose source was in the middle of an operation that disallows tampering with elements or cursors. Either Program_Error is raised, or the operation proceeds as defined.

Implementation Requirements

For each instance of Containers.Hashed_Sets and each instance of Containers.Bounded_Hashed_Sets, if the two instances meet the following conditions then the output generated by the Set'Output or Set'Write subprograms of either instance shall be readable by the Set'Input or Set'Read of the other instance, respectively:

- the Element_Type parameters of the two instances are statically matching subtypes of the same type (note that distinct elaborations of a type declaration result in distinct types); and
- the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of of the two actual Element_Type parameters).

Implementation Advice

Bounded set objects should be implemented without implicit pointers or dynamic allocation.

The implementation advice for procedure Move to minimize copying does not apply.

A.18.24 The Generic Package Containers.Bounded_Ordered_Sets

Insert new clause: [AI05-0001-1; AI05-0160-1; AI05-0184-1]

The language-defined generic package Containers.Bounded_Ordered_Sets provides a private type Set and a set of operations. It provides the same operations as the package Containers.Ordered_Sets (see A.18.9), with the difference that the maximum storage is bounded.

Static Semantics

The declaration of the generic library package Containers.Bounded_Ordered_Sets has the same contents and semantics as Containers.Ordered_Sets except:

- The pragma Preelaborate is replaced with pragma Pure.
- The type Set is declared with a discriminant that specifies the capacity (maximum number of elements) as follows:

```
type Set (Capacity : Count_Type) is tagged private;
```

- The type Set needs finalization if and only if type Element_Type needs finalization.
- If Insert (or Include) adds an element, a check is made that the capacity is not exceeded, and Capacity_Error is propagated if this check fails.
- In procedure Assign, if Source length is greater than Target capacity, then Capacity_Error is propagated.
- The function Copy is replaced with:

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

  Returns a set whose elements are initialized from the values in Source. If Capacity is 0, then the set capacity is the length of Source; if Capacity is equal to or greater than the length of Source, the set capacity is the specified value; otherwise, the operation propagates Capacity_Error.

Bounded Errors

It is a bounded error to use a bounded set if it was the target of an assignment_statement whose source was in the middle of an operation that disallows tampering with elements or cursors. Either Program_Error is raised, or the operation proceeds as defined.

Implementation Requirements

For each instance of Containers.Ordered_Sets and each instance of Containers.Bounded_Ordered_Sets, if the two instances meet the following conditions then the output generated by the Set'Output or Set'Write subprograms of either instance shall be readable by the Set'Input or Set'Read of the other instance, respectively:

- the Element_Type parameters of the two instances are statically matching subtypes of the same type (note that distinct elaborations of a type declaration result in distinct types); and
- the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of of the two actual Element_Type parameters).

Implementation Advice

Bounded set objects should be implemented without implicit pointers or dynamic allocation.

The implementation advice for procedure Move to minimize copying does not apply.

A.18.25 The Generic Package Containers.Bounded_Multiway_Trees

Insert new clause: [AI05-0136-1; AI05-0184-1]

The language-defined generic package Containers.Bounded_Multiway_Trees provides a private type Tree and a set of operations. It provides the same operations as the package Containers.Multiway_Trees (see A.18.10), with the difference that the maximum storage is bounded.

Static Semantics

The declaration of the generic library package Containers.Bounded_Multiway_Trees has the same contents and semantics as Containers.Multiway_Trees except:

- pragma Preelaborate is replaced with pragma Pure.
- The type Tree is declared with a discriminant that specifies the capacity (maximum number of elements) as follows:

```
type Tree (Capacity : Count_Type) is tagged private;
```

- The type Tree needs finalization if and only if type Element_Type needs finalization.
- The allocation of internal storage includes a check that the capacity is not exceeded, and Capacity_Error is raised if this check fails.
- In procedure Assign, if Source length is greater than Target capacity, then Capacity_Error is propagated.
- Function Copy is declared as follows:

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

  If Capacity is 0, then the tree capacity is the count of Source; if Capacity is equal to or greater than Source.Count, the tree capacity equals the value of the Capacity parameter; otherwise, the operation propagates Capacity_Error.

- In the five-parameter procedure Splice_Subtree, if Source is not the same object as Target, and if the sum of Target.Count and Subtree_Count (Position) is greater than Target.Capacity, then Splice_Subtree propagates Capacity_Error.

- In the five-parameter procedure Splice_Children, if Source is not the same object as Target, and if the sum of Target.Count and Subtree_Count (Source_Parent)-1 is greater than Target.Capacity, then Splice_Children propagates Capacity_Error.

Bounded (Run-Time) Errors

It is a bounded error to use a bounded tree if it was the target of an assignment_statement whose source was in the middle of an operation that disallows tampering with cursors or elements. Either Program_Error is raised, or the operation proceeds as defined.

Implementation Requirements

For each instance of Containers.Multiway_Trees and each instance of Containers.Bounded_Multiway_Trees, if the two instances meet the following conditions then the output generated by the Tree'Output or Tree'Write subprograms of either instance shall be readable by the Tree'Input or Tree'Read of the other instance, respectively:

- the Element_Type parameters of the two instances are statically matching subtypes of the same type (note that distinct elaborations of a type declaration result in distinct types); and
- the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of of the two actual Element_Type parameters).

Implementation Advice

Bounded tree objects should be implemented without implicit pointers or dynamic allocation.

The implementation advice for procedure Move to minimize copying does not apply.

A.18.26 Array Sorting

Replace paragraph 1: [AI05-0001-1]

The language-defined generic procedures Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort provide sorting on arbitrary array types.

by:

The language-defined generic procedures Containers.Generic_Array_Sort, Containers.Generic_Constrained_Array_Sort, and Containers.Generic_Sort provide sorting on arbitrary array types.

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: [AI05-0001-1; 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.

The generic library procedure Containers.Generic_Sort has the following declaration:

```
generic
   type Index_Type is (<>);
```

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

Reorders the elements of an indexable structure, over the range First .. Last, such that the elements are sorted in the ordering determined by the generic formal function Before. The generic formal Before compares the elements having the given indices, and the generic formal Swap exchanges the values of the indicated elements. Any exception raised during evaluation of Before or Swap is propagated.

The actual function for the generic formal function Before of Generic_Sort is expected to return the same value each time it is called with index values that identify a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify the elements. The actual function for the generic formal Swap should exchange the values of the indicated elements. If the actual for either Before or Swap behaves in some other manner, the behavior of Generic_Sort is unspecified. How many times the Generic_Sort calls Before or Swap is unspecified.

A.18.27 The Generic Package Containers.Synchronized_Queue_Interfaces

The language-defined generic package Containers.Synchronized_Queue_Interfaces provides interface type Queue, and a set of operations for that type. Interface Queue specifies a first-in, first-out queue.

Static Semantics

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

```
generic
   type Element_Type is private;
package Ada.Containers.Synchronized_Queue_Interfaces is
   pragma Pure(Synchronized_Queue_Interfaces);

   type Queue is synchronized interface;

   procedure Enqueue
     (Container : in out Queue;
      New_Item  : in     Element_Type) is abstract;
   pragma Implemented (Enqueue, By_Entry);

   procedure Dequeue
     (Container : in out Queue;
      Element   :    out Element_Type) is abstract;
   pragma Implemented (Dequeue, By_Entry);

   function Current_Use (Container : Queue) return Count_Type is abstract;
   function Peak_Use (Container : Queue) return Count_Type is abstract;

end Ada.Containers.Synchronized_Queue_Interfaces;
```

```
procedure Enqueue
  (Container : in out Queue;
   New_Item  : in     Element_Type) is abstract;
```

A queue type that implements this interface may have a bounded *capacity*. If the queue object has a bounded capacity, and the number of existing elements equals the capacity, then Enqueue blocks until storage becomes available; otherwise Enqueue does not block. In any case, it then copies New_Item onto the queue.

```
procedure Dequeue
  (Container : in out Queue;
   Element   :    out Element_Type) is abstract;
```

If the queue is empty, then Dequeue blocks until an item becomes available. In any case, it then returns a copy of the element at the head of the queue, and removes it from the container.

```
function Current_Use (Container : Queue) return Count_Type is abstract;
```

Returns the number of elements currently in the queue.

```
function Peak_Use (Container : Queue) return Count_Type is abstract;
```

Returns the maximum number of elements that have been in the queue at any one time.

A.18.28 The Generic Package Containers.Unbounded_Synchronized_Queues

Insert new clause: [AI05-0159-1]

Static Semantics

The language-defined generic package Containers.Unbounded_Synchronized_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue.

```
with System;
with Ada.Containers.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is new Ada.Containers.Synchronized_Queue_Interfaces (<>);
   Default_Ceiling: System.Any_Priority := System.Priority'Last;
package Ada.Containers.Unbounded_Synchronized_Queues is
   pragma Preelaborate(Unbounded_Synchronized_Queues);

   protected type Queue (Ceiling: System.Any_Priority := Default_Ceiling) is
         new Queue_Interfaces.Queue with
      pragma Priority(Ceiling);

      overriding
      entry Enqueue (New_Item: in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue (Element: out Queue_Interfaces.Element_Type);

      overriding
      function Current_Use return Count_Type;
      overriding
      function Peak_Use return Count_Type;

   private
      ... -- not specified by the language
   end Queue;

private

   ... -- not specified by the language

end Ada.Containers.Unbounded_Synchronized_Queues;
```

The type Queue is used to represent task-safe queues.

The capacity for instances of type Queue is unbounded.

A.18.29 The Generic Package Containers.Bounded_Synchronized_Queues

Insert new clause: [AI05-0159-1]

Static Semantics

The language-defined generic package Containers.Bounded_Synchronized_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue.

```
with System;
with Ada.Containers.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is new Ada.Containers.Synchronized_Queue_Interfaces (<>);
   Default_Capacity : Count_Type;
   Default_Ceiling: System.Any_Priority := System.Priority'Last;
package Ada.Containers.Bounded_Synchronized_Queues is
   pragma Preelaborate(Bounded_Synchronized_Queues);

   protected type Queue
        (Capacity : Count_Type := Default_Capacity;
         Ceiling: System.Any_Priority := Default_Ceiling) is
         new Queue_Interfaces.Queue with
      pragma Priority(Ceiling);

      overriding
      entry Enqueue (New_Item: in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue (Element: out Queue_Interfaces.Element_Type);
```

```
      overriding
      function Current_Use return Count_Type;
      overriding
      function Peak_Use return Count_Type;

   private
      ... -- not specified by the language
   end Queue;

private

   ... -- not specified by the language

end Ada.Containers.Bounded_Synchronized_Queues;
```

The semantics are the same as for Unbounded_Synchronized_Queues, except:

- The capacity for instances of type Queue is bounded and specified by the discriminant Capacity.

Implementation Advice

Bounded queue objects should be implemented without implicit pointers or dynamic allocation.

A.18.30 The Generic Package Containers.Unbounded_Priority_Queues

Insert new clause: [AI05-0159-1]

Static Semantics

The language-defined generic package Containers.Unbounded_Priority_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue.

```
with System;
with Ada.Containers.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is new Ada.Containers.Synchronized_Queue_Interfaces (<>);
   type Queue_Priority is private;
   with function Get_Priority
     (Element: Queue_Interfaces.Element_Type) return Queue_Priority is <>;
   with function Before
     (Left, Right : Queue_Priority) return Boolean is <>;
   Default_Ceiling: System.Any_Priority := System.Priority'Last;
package Ada.Containers.Unbounded_Priority_Queues is
   pragma Preelaborate(Unbounded_Priority_Queues);

   protected type Queue
        (Ceiling: System.Any_Priority := Default_Ceiling) is
         new Queue_Interfaces.Queue with
      pragma Priority(Ceiling);

      overriding
      entry Enqueue (New_Item: in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue (Element: out Queue_Interfaces.Element_Type);

      not overriding
      entry Dequeue_Only_High_Priority
        (Low_Priority : in     Queue_Priority;
         Element      :    out Queue_Interfaces.Element_Type);

      overriding
      function Current_Use return Count_Type;
      overriding
      function Peak_Use return Count_Type;

   private
      ... -- not specified by the language
   end Queue;

private

   ... -- not specified by the language

end Ada.Containers.Unbounded_Priority_Queues;
```

The type Queue is used to represent task-safe priority queues.

The capacity for instances of type Queue is unbounded.

Two elements *E1* and *E2* are equivalent if Before(Get_Priority(*E1*), Get_Priority(*E2*)) and Before(Get_Priority(*E2*), Get_Priority(*E1*)) both return False.

Enqueue inserts an item according to the order specified by the Before function on the result of Get_Priority on the elements. If the queue already contains elements equivalent to New_Item, then it is inserted after the existing equivalent elements.

Dequeue_Only_High_Priority is the same as Dequeue, except that it blocks until the element *E* at the head of the queue satisfies Before(Get_Priority(*E*), Low_Priority).

The actual functions for Get_Priority and Before are expected to return the same value each time they are called with the same actuals, and should not modify their actuals. Before should define a strict weak ordering relationship (see A.18). If the actual functions behave in some other manner, the behavior of Unbounded_Priority_Queues is unspecified.

A.18.31 The Generic Package Containers.Bounded_Priority_Queues

Insert new clause: [AI05-0159-1]

Static Semantics

The language-defined generic package Containers.Bounded_Priority_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue.

```
with System;
with Ada.Containers.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is new Ada.Containers.Synchronized_Queue_Interfaces (<>);
   type Queue_Priority is private;
   with function Get_Priority
     (Element: Queue_Interfaces.Element_Type) return Queue_Priority is <>;
   with function Before
     (Left, Right : Queue_Priority) return Boolean is <>;
   Default_Capacity : Count_Type;
   Default_Ceiling: System.Any_Priority := System.Priority'Last;
package Ada.Containers.Bounded_Priority_Queues is
   pragma Preelaborate(Bounded_Priority_Queues);

   protected type Queue
        (Capacity : Count_Type := Default_Capacity;
         Ceiling: System.Any_Priority := Default_Ceiling) is
         new Queue_Interfaces.Queue with
      pragma Priority(Ceiling);

      overriding
      entry Enqueue (New_Item: in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue (Element: out Queue_Interfaces.Element_Type);

      not overriding
      entry Dequeue_Only_High_Priority
        (Low_Priority : in     Queue_Priority;
         Element      :    out Queue_Interfaces.Element_Type);

      overriding
      function Current_Use return Count_Type;
      overriding
      function Peak_Use return Count_Type;

   private
      ... -- not specified by the language
   end Queue;

private

   ... -- not specified by the language

end Ada.Containers.Bounded_Priority_Queues;
```

The semantics are the same as for Unbounded_Priority_Queues, except:

- The capacity for instances of type Queue is bounded and specified by the discriminant Capacity.

Implementation Advice

Bounded priority queue objects should be implemented without implicit pointers or dynamic allocation.

A.18.32 Indefinite Synchronized Queues

Insert new clause: [AI05-0159-1]

There are three generic packages Containers.Indefinite_Synchronized_Queue_Interfaces, Containers.Indefinite_Unbounded_Synchronized_Queues, and Containers.Indefinite_Unbounded_Priority_Queues. These are identical to Containers.Synchronized_Queue_Interfaces, Containers.Unbounded_Synchronized_Queues, and Containers.Unbounded_Priority_Queues, respectively, except except that the generic formal Element_Type is indefinite.

A.19 The Package Locales

Insert new clause: [AI05-0127-2]

A **locale** identifies a geopolitical place or region and its associated language, which can be used to determine other internationalization related characteristics.

Static Semantics

The following language-defined library package exists:

```
package Ada.Locales is
   pragma Preelaborate(Locales);
   pragma Remote_Types(Locales);

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

   Language_Unknown : constant Language_Code := "und";
   Country_Unknown : constant Country_Code := "ZZ";

   function Language return Language_Code;
   function Country return Country_Code;

end Ada.Locales;
```

The *active locale* is the locale associated with the active partition.

Language_Code is a lower-case string representation of an ISO 639-3 alpha-3 code that identifies a language.

Country_Code is an upper-case string representation of an ISO 3166-1 alpha-2 code that identifies a country.

Function Language returns the code of the language associated with the active locale. If the Language_Code associated with the active locale cannot be determined from the environment then Language returns Language_Unknown.

Function Country returns the code of the country associated with the active locale. If the Country_Code associated with the active locale cannot be determined from the environment then Country returns Country_Unknown.

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.6 Shared Variable Control

Replace paragraph 2: [AI05-0009-1]

The form for pragmas Atomic, Volatile, Atomic_Components, and Volatile_Components is as follows:

by:

The form for pragmas Atomic, Volatile, Independent, Atomic_Components, Volatile_Components, and Independent_Components is as follows:

Insert after paragraph 4: [AI05-0009-1]

pragma Volatile(local_name);

the new paragraph:

pragma Independent(*component*_local_name);

Insert after paragraph 6: [AI05-0009-1]

pragma Volatile_Components(*array*_local_name);

the new paragraph:

pragma Independent_Components(local_name);

Replace paragraph 9: [AI05-0009-1]

The local_name in an Atomic or Volatile pragma shall resolve to denote either an object_declaration, a non-inherited component_declaration, or a full_type_declaration. The *array*_local_name in an Atomic_Components or Volatile_Components pragma shall resolve to denote the declaration of an array type or an array object of an anonymous type.

by:

The local_name in an Atomic or Volatile pragma shall resolve to denote either an object_declaration, a non-inherited component_declaration, or a full_type_declaration. The *component*_local_name in an Independent pragma shall resolve to denote a non-inherited component_declaration. The *array*_local_name in an Atomic_Components or Volatile_Components pragma shall resolve to denote the declaration of an array type or an array object of an anonymous type. The local_name in an Independent_Components pragma shall resolve to denote the declaration of an array or record type or an array object of an anonymous type.

Replace paragraph 12: [AI05-0142-4]

If an atomic object is passed as a parameter, then the type of the formal parameter shall either be atomic or allow pass by copy (that is, not be a nonatomic by-reference type). If an atomic object is used as an actual for a generic formal object of mode **in out**, then the type of the generic formal object shall be atomic. If the prefix of an attribute_reference for an Access attribute denotes an atomic object (including a component), then the designated type of the resulting access type shall be atomic. If an atomic type is used as an actual for a generic formal derived type, then the ancestor of the formal type shall be atomic or allow pass by copy.

Corresponding rules apply to volatile objects and types.

by:

If an atomic object is passed as a parameter, then the formal parameter shall either have an atomic type or allow pass by copy. If an atomic object is used as an actual for a generic formal object of mode **in out**, then the type of the generic formal object shall be atomic. If the prefix of an attribute_reference for an Access attribute denotes an atomic object (including a component), then the designated type of the resulting access type shall be atomic. If an atomic type is used as an actual for a generic formal derived type, then the ancestor of the formal type shall be atomic or allow pass by copy. Corresponding rules apply to volatile objects and types.

Insert after paragraph 13: [AI05-0009-1]

If a pragma Volatile, Volatile_Components, Atomic, or Atomic_Components applies to a stand-alone constant object, then a pragma Import shall also apply to it.

the new paragraphs:

It is illegal to apply either an Independent or Independent_Components pragma to a component, object, or type if the implementation cannot provide the independent addressability required by the pragma (see 9.10).

It is illegal to specify a representation aspect for a component, object, or type to which pragma Independent or Independent_Components applies, in a way that prevents the implementation from providing the independent addressability required by the pragma.

Insert after paragraph 14: [AI05-0009-1]

These pragmas are representation pragmas (see 13.1).

the new paragraph:

Pragmas Independent and Independent_Components *specify as independently addressable* the named object or component(s), or in the case of a type, all objects of that type. All atomic objects are considered to be specified as independently addressable.

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 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.2.1 The Task Dispatching Model

Replace paragraph 1.2: [AI05-0166-1]

```
package Ada.Dispatching is
  pragma Pure(Dispatching);
  Dispatching_Policy_Error : exception;
end Ada.Dispatching;
```

by:

```
package Ada.Dispatching is
  pragma Preelaborate(Dispatching);

  procedure Yield;

  Dispatching_Policy_Error : exception;
end Ada.Dispatching;
```

Insert after paragraph 6: [AI05-0166-1]

Each processor also has one *running task*, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point it goes back to one or more ready queues; a task (possibly the same task) is then selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs.

the new paragraph:

A call of Yield is a task dispatching point. Yield is a potentially blocking operation (see 9.5.1).

D.2.4 Non-Preemptive Dispatching

Insert after paragraph 2: [AI05-0166-1]

The *policy_*identifier Non_Preemptive_FIFO_Within_Priorities is a task dispatching policy.

the new paragraphs:

The following language-defined library package exists:

```
package Ada.Dispatching.Non_Preemptive is
   pragma Preelaborate(Non_Preemptive);
   procedure Yield_To_Higher;
   procedure Yield_To_Same_Or_Higher renames Yield;
end Ada.Dispatching.Non_Preemptive;
```

A call of Yield_To_Higher is a task dispatching point for this policy. If the task at the head of the highest priority ready queue has a higher active priority than the calling task then the calling task is preempted.

Replace paragraph 9: [AI05-0166-1]

For this policy, a non-blocking delay_statement is the only non-blocking event that is a task dispatching point (see D.2.1).

by:

For this policy, blocking or termination of a task, a delay_statement, a call to Yield_To_Higher, and a call to Yield_To_Same_Or_Higher or Yield are the only task dispatching points (see D.2.1).

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; AI05-0216-1]

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

by:

No task depends on a master other than the library-level master.

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 5: [AI05-0211-1]

There are no abort_statements, and there are no calls on Task_Identification.Abort_Task.

by:

There are no abort_statements, and there is no use of a name denoting Task_Identification.Abort_Task.

Replace paragraph 10: [AI05-0013-1; AI05-0211-1]

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 a name denoting 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 10.4: [AI05-0211-1]

There are no delay_relative_statements.

by:

There are no delay_relative_statements, and there is no use of a name that denotes the Timing_Events.Set_Handler subprogram that has a Time_Span parameter.

Replace paragraph 10.7: [AI05-0211-1]

There are no calls to the Set_Specific_Handler and Specific_Handler subprograms in Task_Termination.

by:

There is no use of a name that denotes the Set_Specific_Handler and Specific_Handler subprograms in Task_Termination.

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.10 Synchronous Task Control

the new paragraph:

The type Suspension_Object is a by-reference type.

the new paragraphs:

The following language-defined library package exists:

```
with Ada.Real_Time;
package Ada.Synchronous_Task_Control.EDF is
   procedure Suspend_Until_True_And_Set_Deadline
      (S  : in out Suspension_Object;
       TS : in     Ada.Real_Time.Time_Span);
end Ada.Synchronous_Task_Control.EDF;
```

Insert after paragraph 10: [AI05-0168-1]

Program_Error is raised upon calling Suspend_Until_True if another task is already waiting on that suspension object. Suspend_Until_True is a potentially blocking operation (see 9.5.1).

the new paragraph:

The procedure Suspend_Until_True_And_Set_Deadline blocks the calling task until the state of the object S is True; at that point the task becomes ready with a deadline of Ada.Real_Time.Clock + TS, and the state of the object becomes False. Suspend_Until_True_And_Set_Deadline is a potentially blocking operation.

Insert after paragraph 11: [AI05-0168-1]

The implementation is required to allow the calling of Set_False and Set_True during any protected action, even one that has its ceiling priority in the Interrupt_Priority range.

the new paragraph:

NOTES

5 More complex schemes, such as setting the deadline relative to when Set_True is called, can be programmed using a protected object.

D.10.1 Synchronous Barriers

Insert new clause: [AI05-0174-1]

This clause introduces a language-defined package to synchronously release a group of tasks after the number of blocked tasks reaches a specified count value.

Static Semantics

The following language-defined library package exists:

```
package Ada.Synchronous_Barriers is
   pragma Preelaborate(Synchronous_Barriers);

   subtype Barrier_Limit is Positive range 1 .. implementation-defined;

   type Synchronous_Barrier (Number_Waiting : Barrier_Limit) is limited private;

   procedure Wait_For_Release (The_Barrier : in out Synchronous_Barrier;
```

```
                                    Notified    :    out Boolean);

   private
      -- not specified by the language
   end Ada.Synchronous_Barriers;
```

Type Synchronous_Barrier needs finalization (see 7.6).

Dynamic Semantics

Each call to Wait_For_Release blocks the calling task until the number of blocked tasks associated with the Synchronous_Barrier object is equal to Number_Waiting, at which time all blocked tasks are released. Notified is set to True for one of the released tasks, and set to False for all other released tasks.

The mechanism for determining which task sets Notified to True is implementation defined.

Once all tasks have been released, a Synchronous_Barrier object may be reused to block another Number_Waiting number of tasks.

As the first step of the finalization of a Synchronous_Barrier, each blocked task is unblocked and Program_Error is raised at the place of the call to Wait_For_Release.

It is implementation-defined whether an abnormal task which is waiting on a Synchronous_Barrier object is aborted immediately or aborted when the tasks waiting on the object are released.

Wait_For_Release is a potentially blocking operation (see 9.5.1).

Bounded (Run-Time) Errors

It is a bounded error to call Wait_For_Release on a Synchronous_Barrier object after that object is finalized. If the error is detected, Program_Error is raised. Otherwise, the call proceeds normally, which may leave a task blocked forever.

D.13.1 The Ravenscar Profile

Replace paragraph 4: [AI05-0171-1]

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
                No_Abort_Statements,
                No_Dynamic_Attachment,
                No_Dynamic_Priorities,
                No_Implicit_Heap_Allocations,
                No_Local_Protected_Objects,
                No_Local_Timing_Events,
                No_Protected_Type_Allocators,
                No_Relative_Delay,
                No_Requeue_Statements,
                No_Select_Statements,
                No_Specific_Termination_Handlers,
                No_Task_Allocators,
                No_Task_Hierarchy,
                No_Task_Termination,
                Simple_Barriers,
                Max_Entry_Queue_Length => 1,
                Max_Protected_Entries => 1,
                Max_Task_Entries => 0,
                No_Dependence => Ada.Asynchronous_Task_Control,
                No_Dependence => Ada.Calendar,
                No_Dependence => Ada.Execution_Time.Group_Budget,
                No_Dependence => Ada.Execution_Time.Timers,
                No_Dependence => Ada.Task_Attributes);
```

by:

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
                No_Abort_Statements,
                No_Dynamic_Attachment,
                No_Dynamic_Priorities,
                No_Implicit_Heap_Allocations,
                No_Local_Protected_Objects,
```

```
                        No_Local_Timing_Events,
                        No_Protected_Type_Allocators,
                        No_Relative_Delay,
                        No_Requeue_Statements,
                        No_Select_Statements,
                        No_Specific_Termination_Handlers,
                        No_Task_Allocators,
                        No_Task_Hierarchy,
                        No_Task_Termination,
                        Simple_Barriers,
                        Max_Entry_Queue_Length => 1,
                        Max_Protected_Entries => 1,
                        Max_Task_Entries => 0,
                        No_Dependence => Ada.Asynchronous_Task_Control,
                        No_Dependence => Ada.Calendar,
                        No_Dependence => Ada.Execution_Time.Group_Budget,
                        No_Dependence => Ada.Execution_Time.Timers,
                        No_Dependence => Ada.Task_Attributes,
                        No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

Implementation Requirements

A task shall only be on the ready queues of one processor, and the processor to which a task belongs shall be defined statically. Whenever a task running on a processor reaches a task dispatching point, it goes back to the ready queues of the same processor. A task with a CPU value of Not_A_Specific_CPU will execute on an implementation defined processor. A task without a CPU pragma will activate and execute on the same processor as its activating task.

Implementation Advice

On a multiprocessor system, an implementation should support a fully partitioned approach. Each processor should have separate and disjoint ready queues.

D.14 Execution Time

Insert after paragraph 9: [AI05-0170-1]

```
function Time_Of (SC : Seconds_Count;
                  TS : Time_Span := Time_Span_Zero) return CPU_Time;
```

the new paragraphs:

```
Interrupt_Clocks_Supported : constant Boolean := implementation-defined;

Separate_Interrupt_Clocks_Supported : constant Boolean :=
  implementation-defined;

function Clock_For_Interrupts return CPU_Time;
```

Replace paragraph 11: [AI05-0170-1]

The *execution time* or CPU time of a given task is defined as the time spent the system executing that task, including the time spent executing run-time or system services on its behalf. The mechanism used to measure execution time is implementation defined. It is implementation defined which task, if any, is charged the execution time that is consumed by interrupt handlers and run-time services on behalf of the system.

by:

The *execution time* or CPU time of a given task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf. The mechanism used to measure execution time is implementation defined. The Boolean constant Interrupt_Clocks_Supported is set to True if the implementation separately accounts for the execution time of interrupt handlers. If it is set to False it is implementation defined which task, if any, is charged the execution time that is consumed by interrupt handlers. The Boolean constant Separate_Interrupt_Clocks_Supported is set to True if the implementation separately accounts for the execution time of individual interrupt handlers.

Insert after paragraph 13: [AI05-0170-1]

CPU_Time_First and CPU_Time_Last are the smallest and largest values of the CPU_Time type, respectively.

the new paragraph:

The execution time value for the Clock_For_Interrupts is initialized to zero.

Insert after paragraph 18: [AI05-0170-1]

The effects of the Split and Time_Of operations are follows, treating values of type CPU_Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split (T, SC, TS) is to set SC and TS to values such that T*CPU_Time_Unit = SC*1.0 +

TS*CPU_Time_Unit, and 0.0 <= TS*CPU_Time_Unit < 1.0. The value returned by Time_Of(SC,TS) is the execution-time value T such that T*CPU_Time_Unit=SC*1.0 + TS*CPU_Time_Unit.

the new paragraph:

The function Clock_For_Interrupts returns the total cumulative time spent executing within all interrupt handlers. This time is not allocated to any task execution time clock. If Interrupt_Clocks_Supported is set to False the function raises Program_Error.

D.14.2 Group Execution Time Budgets

Replace paragraph 3: [AI05-0169-1]

```
with System;
package Ada.Execution_Time.Group_Budgets is
```

by:

```
with System;
with System.Multiprocessors;
package Ada.Execution_Time.Group_Budgets is
```

Replace paragraph 4: [AI05-0169-1]

```
type Group_Budget is tagged limited private;
```

by:

```
type Group_Budget(P : System.Multiprocessors.CPU :=
                          System.Multiprocessors.CPU'First)
 is tagged limited private;
```

Replace paragraph 21: [AI05-0169-1]

The procedure Replenish loads the group budget GB with To as the Time_Span value. The exception Group_Budget_Error is raised if the Time_Span value To is non-positive. Any execution of any member of the group of tasks results in the budget counting down, unless exhausted. When the budget becomes exhausted (reaches Time_Span_Zero), the associated handler is executed if the handler of group budget GB is set. Nevertheless, the tasks continue to execute.

by:

The procedure Replenish loads the group budget GB with To as the Time_Span value. The exception Group_Budget_Error is raised if the Time_Span value To is non-positive. Any execution on processor P of any member of the group of tasks results in the budget counting down, unless exhausted. When the budget becomes exhausted (reaches Time_Span_Zero), the associated handler is executed if the handler of group budget GB is set. Nevertheless, the tasks continue to execute.

D.14.3 Execution Time of Interrupt Handlers

Insert new clause: [AI05-0170-1]

This clause describes a language-defined package to measure the execution time of interrupt handlers.

Static Semantics

The following language-defined library package exists:

```
with Ada.Interrupts;
package Ada.Execution_Time.Interrupts is
   function Clock (I : Ada.Interrupts.Interrupt_ID)
        return CPU_Time;
   function Supported (I : Ada.Interrupts.Interrupt_ID)
        return Boolean;
end Ada.Execution_Time.Interrupts;
```

The execution time or CPU time of a given interrupt I is defined as the time spent by the system executing interrupt handlers identified by I, including the time spent executing run-time or system services on its behalf. The mechanism used to measure execution time is implementation defined. Time spent executing interrupt handlers is distinct from time spent executing any task.

For each interrupt ID, the execution time value is initially set to zero.

Dynamic Semantics

The function Clock returns the current cumulative execution time of the interrupt identified by I. If Separate_Interrupt_Clocks_Supported is set to False the function raises Program_Error.

The function Supported returns True if the implementation is monitoring the execution time of interrupt I. Otherwise it returns False. For any interrupt ID I for which Supported(I) returns False, the function Clock(I) will return a value equal to

Ada.Execution_Time.Time_Of(0).

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.

Replace paragraph 24: [AI05-0210-1]

An upper bound on the lateness of the execution of a handler. That is, the maximum time between when a handler is actually executed and the time specified when the event was set.

by:

An upper bound on the lateness of the execution of a handler. That is, the maximum time between the time specified for the event and when a handler is actually invoked assuming no other handler or task is executing during this interval.

D.16 Multiprocessor Implementation

Insert new clause: [AI05-0171-1]

This clause allows implementations on multiprocessor platforms to be configured.

Static Semantics

The following language-defined library package exists:

```
package System.Multiprocessors is
   pragma Preelaborate(Multiprocessors);

   type CPU_Range is range 0 .. implementation-defined;
   Not_A_Specific_CPU : constant CPU_Range := 0;
   subtype CPU is CPU_Range range 1 .. CPU_Range'last;

   function Number_Of_CPUs return CPU;
end System.Multiprocessors;
```

A call of Number_Of_CPUs returns the number of processors available to the program. Within a given partition, each call on Number_Of_CPUs will return the same value.

Syntax

The form of a pragma CPU is as follows:

pragma CPU (expression);

Name Resolution Rules

The expected type for the expression is System.Multiprocessors.CPU_Range.

Legality Rules

A CPU pragma is allowed only immediately within a task_definition, or the declarative_part of a subprogram_body. At most one such pragma shall appear within a given construct.

For a CPU pragma that appears in the declarative_part of a subprogram_body, the expression shall be static.

Dynamic Semantics

The expression of a CPU pragma that appears in a task_definition is evaluated for each task object (see 9.1). The CPU value is then associated with the task object whose task_definition contains the pragma.

Elaboration of a CPU pragma given immediately within the declarative_part of a subprogram_body has no effect; the CPU value is not associated with any task unless the pragma occurs within the main subprogram.

The CPU value is associated with the environment task if the pragma appears in the declarative_part of the main subprogram. If a pragma CPU does not apply to the main subprogram it is implementation defined on which processor the environment task executes.

The CPU value determines the processor on which the task will activate and execute; the task is said to be assigned to that

processor. If the CPU value is Not_A_Specific_CPU then the task is not assigned to a processor. A task without a CPU pragma will activate and execute on the same processor as its activating task if the activating task is assigned a processor. If the CPU value is not in the range of System.Multiprocessors.CPU_Range or is greater than Number_Of_CPUs the task is defined to have failed, and it becomes a completed task (see 9.2).

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.2.3 Remote Call Interface Library Units

Replace paragraph 14: [AI05-0101-1]

- it shall not be, nor shall its visible part contain, a subprogram (or access-to-subprogram) declaration whose profile has an access parameter or a parameter of a type that does not support external streaming (see 13.13.2);

by:

- it shall not be, nor shall its visible part contain, a subprogram (or access-to-subprogram) declaration whose profile has a parameter or result of a type that does not support external streaming (see 13.13.2);

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.

by:

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

H.4 Safety and Security Restrictions

Insert after paragraph 8: [AI05-0152-1]

No_Local_Allocators
Allocators are prohibited in subprograms, generic subprograms, tasks, and entry bodies.

the new paragraph:

No_Anonymous_Allocators
Allocators of anonymous access types are not allowed.

Annex J: Obsolescent Features

J.1 Renamings of 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.