Overview
of
Ada 2022
Version 63e

An Ada 2022 Language Enhancement Guide

Jeff Cousins CEng FIET
Member and former chair of the Ada Rapporteur Group

Table of Contents

Chapter 1: Introduction

Let us begin with why do we need another revision, when Ada has been used to successfully deliver so many projects around the world, indeed across the solar system?

The world has moved on. The most significant external factor has probably been the growth of the number of cores on a processor. Making use of the first multi-core processors was relatively easy for Ada compared with other languages as tasking had been included from the outset. Maybe latent timing problems would emerge when tasks were truly running on different processors instead of just pretending to, but these problems were small scale compared with, say, elaboration order problems when moving between different vendors' compilers.

These days a processor may have dozens of cores, maybe in the future it will be hundreds. Thus the first improvement given in the list of instructions (see below) from WG 9 (the ISO/IEC Working Group responsible for Ada), to the Ada Rapporteur Group (ARG), was finer grained control of parallelism.

The ARG follows the following instructions from WG 9, extracted from ISO/IEC JTC 1/SC 22/WG 9 N571:

> "The ARG is requested to pay particular attention to the following two categories of improvements:
>
> 1. Improvements that will maintain or improve Ada's advantages, especially in those user domains where safety and security are prime concerns;
> 2. Improvements that will remedy shortcomings in Ada.
>
> Improvements of special interest in these categories are:
>
> - Improving the capabilities of Ada on multi-core and multi-threaded architectures;
> - Improving the ability to write and enforce contracts for Ada entities (for instance, via preconditions);
> - Improving the use and functionality of the predefined containers;
> - Improving support for Unicode in the language and predefined libraries.
>
> These are all examples of improvements in category A, except for the last which is an example of an improvement in category B."

New real-time features have also arisen, primarily from recommendations from the series of International Real-Time Ada Workshops (IRTAW).

As always, there is the balancing act to be struck between keeping the language stable and backwardly compatible, and adding new features to keep with the times.

Each topic is organized in terms of the relevant Ada Issue(s). When appropriate, a reference to the Reference Manual subclause that primarily defines the feature is also included.

An Ada Issue is an investigation of a comment raised on the Ada standard. These are first worked on and approved by the ARG. They are then passed to WG 9 for consideration and approval before eventually being consolidated and sent to ISO for formal processing to create a revised internation standard. Ada Issues raised on Ada 2012 were assigned numbers of the form AI12-nnnn-aa, where aa is the alternative number. These AIs may be found at http://www.ada-auth.org/AI12-SUMMARY.HTML (for HTML versions of this document, the numbers link to the appropriate AI as well). The relevant AI title and number are quoted in the following sections, omitting the alternative number if there is only one alternative.

Thanks to Randy Brukardt for his many suggestions, and to John Barnes for his review comments, and to the rest of the ARG, especially Tucker Taft, for their examples.

NOTE 1 Contemporary English is used where possible, but where a word is quoted from the RM, then of necessity the US spelling is used.

NOTE 2 RM references give the location of the primary definition of a features; rules defining the feature may occur in other parts of the RM as well. An RM reference is not given when a feature does not have a obvious primary definition; the feature may be defined in multiple places or implicitly in rules defining other concepts.

Chapter 2: Parallelism

Although parallelism may be the first improvement on the list, the parallelism features may be the hardest of the new features to add, so unfortunately they may be the last ones to be implemented.

OpenMP is a promising platform to use for the parallelism features though. It already provides facilities for C, C++ and Fortran. A logical thread of control could be mapped on to an OpenMP lightweight thread. OpenMP has made similar design decisions to Ada 2022. For example, if either an exception is propagated out of a logical thread of control, or there is an explicit transfer of control out of a logical thread of control, then an attempt is made to cancel all logical threads of control within the construct.

Some of the new Ada features are clearly inspired by existing SPARK 2014 features, especially *Global-in and global-out annotations (AI12-0079-3)*, but generalising these to cover the whole of the Ada language, not just the SPARK subset, has been far from trivial.

2.1 Parallel constructs

Parallel operations (AI12-0119) is the prime AI for satisfying the first instruction from WG 9 to the ARG, i.e. "Improving the capabilities of Ada on multi-core and multi-threaded architectures". Parallel block statements and parallel loops are two of the new parallel constructs that are provided.

A parallel block statement (see RM 5.6.1) consists of a set of concurrent activities each specified by a sequence of statements, separated by the reserved word **and**, analogous to the syntax for a select statement where the alternatives are separated by the reserved word **or**.

A parallel loop (see RM 5.5) defines a loop body which is designed such that the various iterations of the loop can run concurrently. The implementation is expected to group the iterations into "chunks" to avoid creating an excessive number of physical threads of control, but each iteration is nevertheless considered for most purposes as its own separate logical thread of control.

Both of these constructs start with the new reserved word **parallel** to clearly indicate that these constructs are designed for parallel execution. The implementation might still not execute the constructs in parallel, but the intent is that if multiple processors are available, some or all of them should be allocated to the execution of the construct.

An example of using a parallel block statement when searching a binary tree:

```
type Expression is tagged null record;
  -- Components will be added by each extension
type Expr_Ptr is access all Expression'Class;
type Binary_Operation is new Expression with
  record
  -- An internal node in an Expression tree
    Left, Right : Expr_Ptr;
  end record;
procedure Traverse (T : Expr_Ptr) is
begin
  -- Recurse down the binary tree
  if T /= null and then
    T.all in Binary_Operation'Class then
    parallel do
      Traverse (T.Left);
    and
      Traverse (T.Right);
    and
      Ada.Text_IO.Put_Line ("Processing " & Ada.Tags.Expanded_Name (T'Tag));
    end do;
  end if;
end Traverse;
```

An example of using a parallel block statement when searching a string for a particular character:

```
function Search (S : String;
                 Char : Character) return Boolean is
begin
  if S'Length <= 1000 then
    -- Sequential scan
    return (for some C of S => C = Char);
  else
    -- Parallel divide and conquer
    declare
      Mid : constant Positive := S'First + S'Length/2 – 1;
    begin
      parallel do
        for C of S(S'First .. Mid) loop
          if C = Char then
            return True;     -- Terminates enclosing do
          end if;
        end loop;
      and
        for C of S(Mid + 1 .. S'Last) loop
          if C = Char then
            return True;     -- Terminates enclosing do
          end if;
        end loop;
      end do;
      -- Not found
      return False;
    end;
  end if;
end Search;
```

An example of using a parallel loop when initialising a two-dimensional Boolean array:

```
parallel
for I in Grid'Range(1) loop
  Grid(I, 1) := (for all J in Grid'Range(2) =>
                   Grid(I,J) = True);
end loop;
```

It is intended that the parallel constructs use lightweight threading so as to incur less overhead than tasking. To reduce implementation complexity and reduce the risk of deadlock, blocking is not allowed in a parallel construct, thus it is a bounded error to invoke an operation that is potentially blocking during the execution of a parallel construct. The compiler may complain if a parallel sequence calls a potentially blocking operation. It may also complain if parallel sequences have conflicting global side-effects.

Whereas *Parallel operations (AI12-0119)* provides the mechanism for iterating in parallel over the elements of an array, *Parallel Container Iterators (AI12-0266)* provides the equivalent mechanism for iterating over containers. The optional reserved word **parallel** may be placed before a **for** statement for its iterator form, not just its loop parameter form. The interfaces in Ada.Containers.Iterator_Interfaces are extended to include two new interfaces, a parallel iterator interface, and a reversible parallel iterator interface.

The **of** form of a parallel for loop should be similar whether it is an array (which can be multi-dimensional) or a container being iterated over, e.g. for a multi-dimensional array:

```
type Matrix is array
      (Integer range <>, Integer range <>) of Real;
Board : Matrix (1 .. 8, 1 .. 8);
parallel
for Element of Board loop
  Element := Element * 2.0;
  -- Double each element of the two-dimensional array Board
end loop;
```

2.2 Nonblocking and data race checks

Data race and non-blocking checks for parallel constructs (AI12-0267), amended by *Revise the conflict check policies to ensure compatibility (AI12-0298)*, provides the rules to check for blocking operations and for race conditions within parallel constructs. A “data race” occurs when two concurrent activities attempt to access the same data object without appropriate synchronization and at least one of the accesses updates the object. Such “conflicting” concurrent activities are considered erroneous. The first AI introduces the notion of Conflict Check policies (see RM 9.10.1), to control the degree of checking for potential data races.

It is important that the default for the new parallel constructs is that all possible conflicts are checked, but for backward compatibility we want the default for tasking constructs to be no checks. Thus the pragma Conflict_Check_Policy permits two separate policies, one for parallel constructs, and one for tasking. The policies for parallel constructs include:

- No_Parallel_Conflict_Checks
- Known_Parallel_Conflict_Checks
- All_Parallel_Conflict_Checks

and similarly the policies for tasking include:

- No_Tasking_Conflict_Checks

- Known_Tasking_Conflict_Checks
- All_Tasking_Conflict_Checks.

The default policy is:

```
pragma Conflict_Check_Policy
      (All_Parallel_Conflict_Checks,
       No_Tasking_Conflict_Checks);
```

If the checking policies for parallel constructs and tasking are the same then the pragma may just take one parameter, thus:

```
pragma Conflict_Check_Policy (No_Conflict_Checks);
```

which is shorthand for:

```
pragma Conflict_Check_Policy
      (No_Parallel_Conflict_Checks,
       No_Tasking_Conflict_Checks);
```

Similarly for the policies Known_Conflict_Checks and All_Conflict_Checks.

Procedural iterator aspects (AI12-0344) adds aspect Parallel_Calls to indicate that a subprogram is intended to be called safely from a parallel context, and asks that this is checked in accordance with the Conflict_Check_Policy that is in effect.

2.3 Defining Nonblocking

Nonblocking subprograms (AI12-0064-2) adds the aspect Nonblocking to Ada (see RM 9.5). This allows specifying the blocking status of a subprogram. If a subprogram is declared to be non-blocking, the Ada compiler will attempt to verify that it does not execute any potentially blocking operations. Potentially blocking operations include **select**, **accept**, **entry** call, **delay** and **abort** statements, creating or activating a task, or calling something else which in turn executes one of these.

The language also includes some deadlocks in the definition of "potentially blocking", mostly to simplify the description of the language (deadlocks have nothing to do with blocking). These cases cannot be detected at compile time, but the run-time check of pragma Detect_Blocking can be used if it is desired to detect these problems.

Note that the Nonblocking aspect may be specified for generic formal parameters.

Rather than specify the Nonblocking aspect for many individual subprograms, it can be specified for a **package**, **protected type**, **task** or **generic**, in which case it sets the default for everything within. Indeed for a **protected type**, it may not be specified for individual operations within.

The Nonblocking aspect is fixed as False for an **entry**, and as True for a predefined operator of an elementary type, as one might expect.

Contracts for container operations (AI12-0112) then uses the Nonblocking aspect to specify the non-blocking status for the predefined Containers.

Specifying Nonblocking for Language-Defined Units (AI12-0241) then uses the Nonblocking aspect to specify the non-blocking status for the remainder of Ada's own units (that is, child units of packages Ada, System and Interfaces).

Nonblocking for Unchecked_Deallocation is wrong (AI12-0319) tidies up some of the details of the Nonblocking aspect. It may seem strange that a type can have blocking properties, not just subprograms having them, but an object coming into or out of existence is not always a passive affair. If it is of a

controlled type, then it may have Initialize, Adjust and Finalize procedures; there might be default initialisation of record components, which could call a function; and if on the heap there will be Allocate, Deallocate, and Storage_Size subprograms. For a type to be non-blocking, whichever of these are applicable need to be non-blocking too. (Default initialisation of scalars is not of concern as the Default_Value aspect may only have a static value.)

The standard storage pool(s) are defined to be non-blocking. For a user-defined storage pool to be non-blocking, its Allocate, Deallocate, and Storage_Size subprograms must also be non-blocking. The Unchecked_Deallocation generic invokes a Finalize procedure, so for an instance of it to be non-blocking it must be instantiated with a non-blocking type.

Fixes for Nonblocking (AI12-0374-2) clarifies what happens for generic instantiations. At the point of instantiation, the Nonblocking aspects of the actual generic parameters are **and**-ed with the Nonblocking aspects of the operations within the generic. Thus a non-blocking generic can be instantiated with blocking actuals, in which case the instance will allow blocking. If the instance is required to be non-blocking, then the specific instance can be declared as such.

Also, the Nonblocking aspect may be specified on subtypes, not just types, so that predicates on some subtypes of a given type may call a blocking operation and predicates on some other subtypes of the type may not.

The Nonblocking aspect should also account for preconditions, postconditions, predicates and type invariants applicable to the call of the subprogram.

AI12-0374-2 also removes the notion of a Nonblocking attribute, which had (temporarily) been added by AI12-0064-2.

Fixups for Global annotations (AI12-0380) (which, despite the name, is mostly applicable to both the Nonblocking aspect and the Global annotations) provides finer grain control regarding the use of generic formal parameters and dispatching calls, in optional Annex H for High Integrity Systems.

It is common, especially in larger generic packages, for some of the operations to depend on few or none of the generic parameters. However, normally entities declared within a generic unit are presumed to make use of all the generic formal parameters and the effective Nonblocking aspect and Global annotation reflect this. In order to better describe the use for formal parameters, this AI adds aspect Use_Formal (see RM H.7.1) followed by a list of which generic formal parameters are actually used (enclosed by round brackets and comma-separated if more than one), the reserved word **null** for none, or the reserved word **all** for all of them (which is the default anyway).

For example, in the containers packages, there are many operations (such as First, Next and Length) which simply depend on the state of a container object and do not depend on any of the generic formal parameters. For such operations, we can use

```
with Nonblocking, Global => null, Use_Formal => null,
```

and be assured that these operations will always be nonblocking and never depend on any global objects regardless of the actual parameters for an instance. This means that such operations can be used inside of a parallel operation no matter what the conflict detection state is (see 2.2).

Note that operations that create or copy elements depend on the Element_Type formal parameter, which might allow blocking or depend on global objects (via Initialize, Adjust or Finalize). Such operations cannot use Use_Formal => **null** and must either omit the Use_Formal aspect or use Use_Formal => Element_Type.

Some operations may be implemented by dispatching calls. Normally dispatching calls are checked using the applicable *Nonblocking and* Global'Class aspects. Typically, the Nonblocking aspect and Global

annotation for such calls are very conservative, as one does not want to prevent complex implementations in extensions (even ones to be designed in the future). In particular, most such calls allow blocking, as blocking can be useful in some circumstances, and even the absense of blocking in existing extensions does not prevent it from being added later. (Additionally, the default is to allow blocking for compatibility reasons.)

Often a caller will know more about the routine actually called in a dispatching call, as the type of the controlling object will be known. This AI adds aspect Dispatching (see RM H.7.1) followed a list of dispatching calls (enclosed by round brackets and comma-separated if more than one, and each followed by the name of an object also enclosed by round brackets) that may potentially be called, for which the caller of the original subprogram will account for the blocking state and any globals accessed.

An example of using the Dispatching aspect to manage the globals accessed is found in 2.4, “Defining access to global data”.

2.4 Defining access to global data

Global-in and global-out annotations (AI12-0079-3) (see RM 6.1.2) allow the programmer to specify what global data a subprogram uses, in a manner that is similar to that by which subprogram parameters are specified. Specifying the "side effects" (i.e. effects other than via a parameter) of a subprogram makes it easier for static analysis tools to reason, and (especially when there are no side-effects) can be useful to the human reader as well. For example:

```
type Operating_Mode_Type is
    (Initialising, Normal, Fallback, Shutting_Down);
type Status_Type is (Success, Inaccurate, Failed);

Data_Table     : ...;
Operating_Mode : Operating_Mode_Type;
Status         : Status_Type;

procedure Process_Data_Table
   with
     Global => (in     Operating_Mode;
                out    Status;
                in out Data_Table);
```

This should be fairly familiar to SPARK users. Since we are extending the language proper, we can use the reserved words **in**, **out** and **in out** rather than the SPARK 2014 terms Input, Output and In_Out. Note though that, unlike SPARK 2014, there is no inner => after each mode.

The Global'Class aspect can be specified for a dispatching subprogram, giving an upper bound on the set of global variables that any subprogram dispatched to may access.

For each mode there can be a list of global variables (comma-separated if more than one), the reserved word **all** for all global variables, or the reserved word **synchronized** for all synchronized variables (i.e. tasks, protected objects and atomic objects, the implication being that accesses to them are thread-safe). (*Meaning of Global when there is no mode (AI12-0375)* tweaks the syntax to use semicolons to separate the list of variables for each mode as in the examples above).

The intention is that although advanced users may impose stricter requirement on themselves, the typical user should have to specify few, if any, global aspects. Thus the global aspect for a library unit usually defaults to "Unspecified", i.e. read and write of an unspecified set of global variables (sounds a bit like Rumsfeld's "known unknowns"!), although to **null** for Pure library units, i.e. no read or write of any global variable. For other entities, the global aspect defaults to that of the enclosing library unit.

Besides covering any global variables accessed by the body of the subprogram, the global aspect should also cover those accessed by any preconditions, postconditions, predicates and type invariants. Global variables accessed by other subprograms that the subprogram calls should normally also be identified, though if the other subprogram is passed in as an access-to-subprogram parameter then it is up to the caller of the original subprogram to take account of the effects of whatever subprogram it passes in. If an access-to-variable value is created then presumably the variable that it designates is going to be written, and if an access-to-constant value is created then presumably the constant that it designates is going to be read, so these accesses should be identified too. Generic formal parameters do not need to be included in the global aspect, as the globals used by the actual parameters of an instantiation are automatically added to the globals used by each entity declared by the instance. However, the core language does not check accesses to objects reached via dereferences of access values (the expectation being that such checks are provided by implementation-defined mechanisms).

Optional Annex H, for High Integrity Systems, adds restriction No_Unspecified_Globals, disallowing the Global and Global'Class for a library-level entity from being set or defaulting to Unspecified, thereby forcing the specification of Global. It also adds the restriction No_Hidden_Indirect_Globals, requiring that any accesses to objects reached via dereferences of access values are identified. For example:

```
package P is
  type G is private;
  type Ref (Data : access T) is null record;
  Glob : G;
  ...
  function F (C : aliased in out Container;
                  Pos : Cursor) return Ref
    with Global => in Glob;
  ...
private
  type G is record
    Info : access T;
  end record;
end P;

package body P is
  ...
  function F (C : aliased in out Container;
                  Pos : Cursor) return Ref is
  begin
    return Ref'(Data => Glob.Info);
      -- Error!
      -- The above returns a writable reference to
      -- Glob.Info.all, but Glob is of mode in, and
      -- Glob.Info.all is reachable from Glob.
  end F;
  ...
end P;
```

Optional Annex H allows the global aspect to be specified for subtypes and certain generic formal parameters.

Optional Annex H also provides an extension for dealing with “handles”, for example the File_Type of Text_IO or the Generator of Discrete_Random. Hence, in:

```
procedure Put (File : in File_Type; Item : in String);
```

the File parameter is of mode **in** as the parameter itself isn't modified, yet the state associated with the file is modified. This can now be indicated using an overriding global mode, thus:

```
procedure Put (File : in File_Type; Item : in String)
  with Global => overriding in out File;
```

Fixups for Global annotations (AI12-0380) provides finer grain control regarding the use of generic formal parameters and dispatching calls, in optional Annex H. See the preceding section, 2.3, “Defining Nonblocking” for details.

Control over dispatching calls can be very useful in cases such as stream attributes. For example:

```
type T is tagged private
  with Input => Stream_Input;
procedure Fill (X : out T'Class;
          Str : aliased in out Ada.Streams.Root_Stream_Type'Class)
  with Global => in Debug,
       Dispatching => (T'Input (X), Display (X), Read (Str));
.. -- That was the spec of Fill; the body is below
procedure Fill (X : out T'Class;
          Str : aliased in out Ada.Streams.Root_Stream_Type'Class) is
begin
  X := T'Input (Str'Access);
  if Debug then
    Display (X);
  end if;
end Fill;
```

Without the Dispatching aspect, Fill would have to allow blocking and be assumed to write to all global objects. With it in place, the compiler can examine the type of X and Str to determine the actual operations invoked to determine the blocking state and global objects accessed by a specific call on Fill.

Contracts for container operations (AI12-0112) then uses the global annotations mechanism to specify the Global aspect for the predefined Containers.

Default Global aspect for language-defined units (AI12-0302) then uses the global annotations mechanism to specify the Global aspect for the remainder of Ada's own units (that is, child units of packages Ada, System and Interfaces).

For most language-defined packages (that are not Pure) an explicit value of "**in out synchronized**" (i.e. read and write of the set of global variables that are tasks, protected objects, or atomic objects) is added.

But where some unknown, unsynchronised variable holds state (such as Current_Input or Current_Output for Text_IO) then only "**in out all**" can be stated. This would mean that two concurrent subprogram calls using either Current_Input or Current_Output would be considered to conflict.

Some parameters may be “handles”, for example the File_Type of Text_IO, which even if of mode **in** may be used by a subprogram to update state. For these the value will be "**overriding in out** <param>".

2.5 Reduction

Reduction expressions are another new form of expression, added by Ada 2022, on top of those already added by Ada 2012 (e.g. if expressions, case expressions, quantified expressions). Before the parallel forms can be described, the sequential forms have to be described. These make use of *Container aggregates; generalized array aggregates (AI12-0212)* (see 4.5), which in turn makes use of *Index parameters in array aggregates (AI12-0061)* (see 7.4), so first a preview of them.

Index parameters in array aggregates (AI12-0061) allows a loop parameter to be used in an array aggregate, for example:

```
subtype Index_Type is Positive range 1 .. 10;
type Array_Type is array (Index_Type) of Positive;
Squares_Array : Array_Type := (for I in Index_Type => I * I);
```

Container aggregates; generalized array aggregates (AI12-0212) introduces container aggregates for initialising containers, using square brackets not round brackets (parentheses), and allowing square brackets as an alternative to round brackets for array aggregates.

Iteration is possible within the container aggregate, for example to create a set whose elements all have double the value of the corresponding elements of another set:

```
Doubles_Set : My_Set := [ for Item of X => Item * 2 ];
```

Map-Reduce attribute (AI12-0262) provides a mechanism (see RM 4.5.10) to take a stream of values – a "value sequence" – from an aggregate, and repeatedly apply the same operation to combine the values to produce a single result. Examples include adding a sequence of squares for "sum of squares" or multiplying a sequence of numbers when calculating a factorial. An initial value is required, usually something neutral that has no effect on the result, such as 0 for addition or 1 for multiplication. A parallel version is provided, though if the combining operation is something simple such as addition then the overhead of managing the parallelism is likely to outweigh any performance benefit of performing the additions in parallel. Some examples:

```
-- A reduction expression that outputs the sum of squares
Put_Line ("Sum of Squares is" & Integer'Image
  ([ for I in 1 .. 10 => I**2 ]'Reduce("+", 0));
-- An expression function that returns its result as
-- a Reduction Expression
function Factorial (N : Natural) return Natural is
  ([ parallel for J in 1..N => J ]'Reduce("*", 1));
```

It is important to note that the values are not put in some temporary array then combined, but are combined "on the fly" as each value is produced.

Shorthand Reduction Expressions for Objects (AI12-0242) provides a shorthand for cases where the object is an array or iterable container. For example:

```
Sum : constant Integer := A'Reduce("+", 0);
```

is short for:

```
Sum : constant Integer :=
    [ for Value of A => Value ]'Reduce("+", 0);
```

Similarly:

```
Sum : constant Integer := A'Parallel_Reduce("+", 0);
```

is short for:

```
Sum : constant Integer :=
    [ parallel for Value of A => Value ]'Reduce("+", 0);
```

2.6 Control over the degree of parallelism

Explicit chunk definition for parallel loops (AI12-0251) gives the user control of the degree of parallelism, for example if processing 100 elements of an array on a 20 core machine one may wish to

have 10 logical threads of control (potentially executing on one core each) each processing a group of 10 elements (leaving 10 cores free for other things). Such a group is referred to as a “chunk”.

The optional chunk specification (see RM 5.5) is placed, enclosed by round brackets, after the reserved word **parallel**.

In the more complicated form, an identifier is given that can be used within the parallel construct, for instance:

```
declare
   Partial_Sum : array (1 .. Max_CPUs_To_Use) of Integer := (others => 0);
begin
   parallel (Chunk in Partial_Sum'Range)
   for I in Arr'Range loop
      declare
         Z : Integer;
      begin
         Z := some_complicated_computation;
           ... -- Complex machinations
         Partial_Sum (Chunk) := @ + Arr (I)**2 * Z;
           ... -- Other stuff that also happens in this
           ... -- very complicated loop ...
      end;
   end loop;
   Sum := Partial_Sum'Reduce ("+", 0);
end;
```

This makes uses of a reduction expression, as described above, and @ is a shorthand for the left hand side (see 7.7 in the Others section).

In the simpler form, just the maximum number of logical threads of control to be created to execute the loop is given:

```
parallel (Max_CPUs_To_Use)
for I in Arr'Range loop
   A (I) := B (I) + C (I);
end loop;
```

2.7 An example of the features working together

Not directly related to parallelism, *Index parameters in array aggregates (AI12-0061)* is also used in the example below as it is just such a useful new feature.

```
-- AI12-0241 Specifying Nonblocking for Language-Defined Units
-- AI12-0079-3 Global-in and global-out annotations –
--   default Global => null (i.e. no read or write of any
--   global variable) for Pure packages
-- package Ada.Numerics.Generic_Elementary_Functions
--    with Pure, Nonblocking is
--    function Sqrt (X : Float_Type'Base) return Float_Type'Base;
--    ...
with Ada.Numerics.Elementary_Functions;
-- ...
declare
   Max_CPUs_To_Use : constant := 10;
   Max             : constant := 100;
   subtype Range_Type is Positive range 1 .. Max;
   type Float_Array_Type is array (Range_Type) of Float;
   type Positive_Array_Type is array (Range_Type) of Positive;
   -- AI12-0061 Index parameters in array aggregates
   -- modified by Container aggregates; generalized array
   -- aggregates (AI12-0212) to use [ ]
   Numbers : constant Positive_Array_Type :=
               [ for I in Range_Type => I ];
   Squares       : Positive_Array_Type;
   Square_Roots  : Float_Array_Type;
   Sum_Of_Squares : Integer;
   -- AI12-0064-2 - Nonblocking subprograms
   -- AI12-0079-3 Global-in and global-out annotations
   function Square (P : Positive) return Positive is (P**2)
     with Nonblocking, Global => null;
begin
   -- Ignoring any risk of overflows...

   -- AI12-0119 Parallel operations
   -- Iteration over the elements of an array
   -- AI12-0251-1 Explicit chunk definition for parallel loops
   parallel (Max_CPUs_To_Use)
   for I in Range_Type loop
     Squares      (I) := Square (I);
     Square_Roots (I) := Ada.Numerics.Elementary_Functions.Sqrt (Float (I));
   end loop;
   -- AI12-0242 Shorthand Reduction Expressions for Objects
   -- (dependent on AI12-0262 Map-Reduce attribute)
   Sum_Of_Squares := Squares'Reduce ("+", 0);
end;
```

Chapter 3: Contracts

Ada 83 introduced the generic contract model, whereby a contract is imposed on the types that can be used to instantiate a unit. Parameter modes, and subtypes with constraints, also dating from Ada 83, can be regarded as forms of contract. Ada 2012 added new forms: preconditions, postconditions, type invariants and subtype predicates. Ada 2022 adds further forms: aspect Nonblocking states that no potentially blocking operation should be called; global-in and global-out annotations to describe the use of global objects. Ada 2022 also offers improvements to the existing forms of contract. One of the key benefits of contracts is that they allow checking by static analysis tools.

3.1 Nonblocking contracts

This form of contract is covered in the Parallelism section (see 2.3) since the driving reason for adding it was safe parallelism.

3.2 Global data contracts

This is also covered in the Parallelism section (see 2.4) as again the driving reason for adding it was safe parallelism, but it is more generally useful, for dataflow analysis, for example, or simply documenting behaviour that some coding standards would ask for in a comment.

3.3 Contracts for container operations

Container operation contracts (AI12-0112) specifies the checks to be performed upon entry to a container's subprograms. These are now expressed using Ada 2012 preconditions rather than English.

Checking of the preconditions by the containers' users can be suppressed using:

```
pragma Suppress (Containers_Assertion_Check);
```

The aspects Nonblocking (AI12-0064-2). and Global (AI12-0079-3). are included in the contracts. Some postconditions are added for "easy" results (such as a requirement that a parameter be empty on return). Postconditions on language-defined operations are required to succeed (otherwise the implementation is wrong! – AI12-0179).

3.4 Delta aggregates

Partial aggregate notation (AI12-0127) introduces a new syntactic form of aggregate, the delta aggregate (see RM 4.3.4). This allows one to update one or more fields of a composite object without having to specify every field. This will be particularly useful for postconditions, where one might want to check that only certain fields of a composite parameter had changed, for example:

```
procedure Twelfth (D : in out Date)
  with Post => D = (D'Old with delta Day => 12);
```

The values of the Year and Month components of the delta aggregate are the same as those of D'Old but the Day component is 12.

Delta aggregates always require the changed components to be written in the form of named associations. (Recall that an aggregate is made up of a list of associations; a named association includes an explicit specification of the component(s) to specify, the choose symbol (=>), and the expression giving the

component value. For an array aggregate, the component specification is usually a discrete choice list, often referred to as the choices of the association.)

Unlike other aggregates, there is not a completeness check, as the whole point of a delta aggregate is to give some but not all components.

For record delta aggregates, most other rules are the same as for other record aggregates. Components can only be given once and are evaluated in an arbitrary order. A discriminant may not be specified as a component as to change a discriminant without changing its dependent components could prove disastrous. If a component is in a variant part, then a Discriminant_Check is performed as for the use of a component.

Array delta aggregates are more different than other array aggregates. There are a few limits on index expressions (the main one being that non-static expressions have to be the only one of an association, but multiple associations are allowed even when some are non-static). Unlike other aggregates, such as record delta aggregates, the components are evaluated in the order given. A component may be given multiple times, in which case the value from the last occurrence is the one used.

Delta aggregates can be used with target name symbols (see 7.7) to simplify setting several components at a time. For instance, consider the following Ada 2012 code to calculate basic statistical information for the parent of a particular tree node:

```
Node.Parent.Count  := Node.Parent.Count + 1;
Node.Parent.Sum    := Node.Parent.Sum + Value;
Node.Parent.Sq_Sum := Node.Parent.Sq_Sim + Value*Value;
```

In Ada 2022, one could instead write:

```
Node.Parent := (@ with delta
                    Count => @.Count + 1,
                    Sum => @.Sum + Value,
                    Sq_Sum => @.Sq_Sum + Value*Value);
```

This makes it clear that the intent is to update all of these components as a group without touching any other components (in particular, not changing the position of the node in the tree). It also is safer than the original code, as the delta aggregate does not allow setting the same component twice by mistake, and the target name symbol eliminates misspellings in the name of the node (see 7.7).

3.5 Stable properties

Stable properties of abstract data types (AI12-0187) adds the new aspect Stable_Properties (see RM 7.3.4) to simplify the description of properties of an abstract data type (ADT), by making it easy to specify properties that are usually unchanged by most of the operations of the ADT. The classic example is the Mode of a file, which is unchanged by all of the operations other than Create/Open/Close/Reset (and Set_Mode for streams). The stable properties are automatically included in the postconditions of all the primitive operations of the ADT, decreasing clutter and increasing the information that provers can use.

The aspect Stable_Properties can be given on a partial view or on a full type with no partial view, and also on primitive subprograms. The subprogram version can be used to override the type version when necessary for specific primitive subprograms of the type. The aspect is also applicable to class-wide versions.

The aspect is followed by a list of the stable property functions (comma-separated if more than one) for the primitive subprogram(s). *Syntax for Stable_Properties aspects (AI12-0285)* tweaks the syntax so that

the list is enclosed by round brackets, in the form of a positional aggregate, to avoid possible ambiguity in a list of aspects.

The postcondition(s) of the subprogram are modified with an item that verifies that the property is unchanged for each parameter of the appropriate type, unless that property is already referenced in the explicit postcondition (or inherited postcondition, in the case of class-wide postconditions).

To expand on the example of the Mode of a file, suppose that we wish many subprograms to behave as if they have a postcondition as in:

```
procedure Put (File : in File_Type; Str : in String)
  with Pre  => Mode(File) /= In_File,
       Post => Mode(File) = Mode(File)'Old;
```

Then rather than having to repeat this postcondition for numerous subprograms, if Ada.Text_IO could be rewritten in the form:

```
package Ada.Text_IO is
  type File_Type is private
    with Stable_Properties => (Is_Open, Mode);
...
```

then the declaration of Put could simply be:

```
procedure Put (File : in File_Type; Item : in String)
  with Pre => Mode(File) /= In_File;
```

since we have stated that the Mode is a stable property.

3.6 Declare expressions

Declare expressions (AI12-0236) (see RM 4.5.9). As the power of expressions has grown, some felt that it would help to allow local constants and object renamings within an expression, to avoid repeated subexpressions. For example, the postcondition for Fgetc could be clarified from:

```
(if Stream.The_File (Stream.Cur_Position'Old) =
  EOF_Ch
 then Stream.Cur_Position = Stream.Cur_Position'Old
   and then Result = EOF
 elsif Stream.The_File (Stream.Cur_Position'Old) =
   ASCII.LF
 then Stream.Cur_Position = Stream.Cur_Position'Old
   and then Result = Character'Pos (ASCII.LF)
 else
   Stream.Cur_Position = Stream.Cur_Position'Old + 1
   and then Result = Character'Pos (Stream.The_File
    (Stream.Cur_Position'Old)))
```

to:

```
(declare
   Old_Pos : constant Position :=
             Stream.Cur_Position'Old;
   The_Char : constant Character :=
             Stream.The_File(Old_Pos);
   Pos_Unchg : constant Boolean :=
             Stream.Cur_Position = Old_Pos;
 begin
   (if The_Char = EOF_Ch
     then Pos_Unchg and then Result = EOF
    elsif The_Char = ASCII.LF
     then Pos_Unchg and then Result =
                Character'Pos(ASCII.LF)
    else
     Stream.Cur_Position = Old_Pos + 1
     and then Result = Character'Pos (The_Char)))
```

This uses the reserved words **declare** and **begin**, as for a block, but not **end**, as it is only used within parentheses. *Declare expressions can be static (AI12-0368)* allows these new declare expressions to be static.

3.7 Other support for pre- and postconditions

Pre/Post for access-to-subprogram types (AI12-0220) allows Pre and Post aspects for access-to-subprogram types, so that contract information is available when calling a subprogram indirectly via an access value, as well as (or even instead of) when called directly. For example, to check that a parameter is even:

```
type T1 is access procedure (X : Integer)
   with Pre => X mod 2 = 0;
procedure Foo (X : Integer) is ... end;
...
   Ptr1 : T1 := Foo'Access;
begin
   Ptr1.all (222); -- Precondition check performed
```

Making 'Old more flexible (AI12-0280-2). Currently the following is illegal as the A.**all** in A.**all**'Old is "potentially unevaluated":

```
procedure Proc (A : access Integer)
   with Post =>
     (if A /= null then (A.all'Old > 1 and A.all > 1));
```

This is now relaxed. Thinking less of what it may not be possible to evaluate, but more of what *can* be evaluated in advance, the term "known on entry" is introduced to cover such expressions (the most obvious example being a static expression), and if it is possible to tell on entry to a subprogram that an X'Old need not be evaluated then it isn't. In the example, A is an **in** parameter of an elementary type (which includes access types) so it is passed by copy and cannot change, so if A is **null** on entry then A.**all**'Old would not be evaluated.

3.8 Aspects for Generic Formal Parameters

Previously, language-defined aspects were not allowed for generic formal parameters, but now several are allowed:

As mentioned in the Parallelism section (see 2.3), *Nonblocking subprograms (AI12-0064-2)* allows the Nonblocking aspect to be specified for generic formal parameters, and *Fixes for Nonblocking (AI12-0374-2)* clarifies that, at the point of instantiation, the Nonblocking aspects of the actual generic parameters are **and**-ed with the Nonblocking aspects of the operations within the generic.

The new aspect for types, Default_Initial_Condition (*Default_Initial_Condition for types (AI12-0265)* – see 3.10) is allowed on generic formal private types.

Contracts for generic formal parameters (AI12-0272) allows Pre and Post on generic formal (nonabstract) subprograms. For example:

```
generic
  type Foo is ...
  with function Reduce (Obj : Foo) return Integer
    with Post => Reduce'Result in -9 .. 9;
package Gen is
  ...
end Gen;
```

These conditions are "add"-ed to those for the actual subprogram.

Atomic, Volatile, and Independent generic formal types (AI12-0282). The aspects Atomic, Volatile, Independent, Atomic_Components, Volatile_Components, and Independent_Components can now be specified for generic formal types. The actual type must have a matching specification, though for backward compatibility reasons the actual types can be Atomic, etc., without the formal types necessarily matching.

3.9 Defaults for generic formal types

Defaults for generic formal types (AI12-0205) provides easier and more natural generic instantiation. It uses the reserved words **or use** (see RM 12.5). For example:

```
generic
  type Item_Type is private;
  type Item_Count is range <> or use Natural;
            -- New syntax using or use
  with function "=" (L, R : in Item_Type) return Boolean;
package Lists is
  ...
end Lists;
```

This allows the instantiator to be able to provide a type for the Item_Count, but it can simply be omitted in ordinary circumstances (in which case Natural would be used).

3.10 Default_Initial_Condition for types

Default_Initial_Condition for types (AI12-0265) introduces a new contract aspect, Default_Initial_Condition (see RM 7.3.3), which may be specified for a private type (or private

extension). As with other contracts, it tells the reader and analysis tools what to expect, in this case what the properties of a default initialised object of a type should be.

After the successful default initialisation of an object of the type, a default initial condition check is performed. Since the behaviour of this check does not depend on how the private type is used, the check can only fail if the implementation (or the contract) is incorrect. In the case of a controlled type, the check is performed after the call to the type's Initialize procedure. For example:

```
package Sets is
  type Set is private
    with Default_Initial_Condition => Is_Empty (Set);
  function Is_Empty (S : Set) return Boolean;
  ...
end Sets;
```

3.11 Aspect No_Return for functions

The aspect No_Return may now be specified for functions (*Aspect No_Return for functions (AI12-0269)*, not just procedures (see RM 6.5.1), but the reason that such a function never returns must be that it raises an exception (rather than containing an endless loop). As for procedures, there will be a check at compile time that the function does not contain any explicit return statements, and a check at run-time that it does not run into the final end.

Chapter 4: Containers and Iterators

4.1 Stable Containers to reduce tampering checks

Stable Containers to reduce tampering checks (AI12-0111) attempts to address performance concerns about Ada containers, whilst maintaining their safety. Each container is given a nested package named Stable. This has similar contents to the parent package, but provides a variant of the container type that cannot grow or shrink, and omits operations that might tamper with elements of the container. Such a container can be created by calling the Copy function, or by creating a stabilised view of a normal container. The operations of the Stable package do not perform tampering checks as they are not needed, because the operations that tamper with elements have been omitted. The tampering checks are considered to incur the main performance overhead. The **of** form of a loop will automatically use the stable form of a container within the loop, eliminating the need to set tampering more than once and eliminating the need for most tampering checks within the loop.

4.2 Container operation contracts

As described in the Contracts section (*Container operation contracts (AI12-0112)* – see 3.3), expresses the checks to be performed upon entry to a container's subprograms using Ada 2012 preconditions rather than English.

The aspects Nonblocking (AI12-0064-2 – see 2.3) and Global (AI12-0079-3 – see 2.4) are included in the contracts.

The new aspect Allows_Exit (see *4.4, "Loop-body as anonymous procedure" — AI12-0189* below) is applied to the container Iterate procedures.

All container operations now have versions with the container as the first parameter. This allows a more specific global contract to be specified. Thus:

```
function Next (Position : Cursor) return Cursor
  with Nonblocking, Global => in all, ...
```

but:

```
function Next (Container : Vector; Position : Cursor)
 return Cursor
  with Nonblocking, Global => null, ...
```

The former has to allow anything to be read as the container is not explicitly identified, whereas the latter does not access any globals. The two parameter version also has the benefit of allowing prefix notation to be used. The new versions include Element, Query_Element, Next and Previous for vectors and lists; Key, Element, Query_Element, Next and (if ordered) Previous for maps; Element, Query_Element, Next and (if ordered) Previous for sets; and Subtree_Node_Count, Element, Query_Element, Next_Sibling and Previous_Sibling for multiway trees. If the cursor points to a different container to the one given, then Program_Error is raised.

Consider the following example:

```
package String_Indexes is
  new Indefinite_Hashed_Maps (Key_Type => String,
 ...
  My_Index : constant String_Indexes.Map := String_Indexes.Empty_Map;
function Long_Strings_Count
           (The_Index  : String_Indexes.Map;
            Min_Length : Positive) return Natural is
  Count : Natural := 0;
begin
  for My_Cursor in The_Index.Iterate loop
    if String_Indexes.Key (Position =>
                    My_Cursor)'Length >= Min_Length then
      Count := Count + 1;
    end if;
  end loop;
  return Count;
end Long_Strings_Count;
```

Note the incongruous mixing of OO-style prefix notation in The_Index.Iterate and traditional notation in String_Indexes.Key. In Ada 2022 the latter can be replaced by The_Index.Key, thus allowing the consistent use of prefix notation.

4.3 Use subtype_indication in generalized iterators

Ada 2012 added the ability to simplify:

```
Vec : Int_Vectors.Vector;
...
for I in Vec.Iterate loop
  Vec(I) := Vec(I) + 1;
end loop;
```

to:

```
Vec : Int_Vectors.Vector;
...
for E : T of Vec loop
   E := E + 1;
end loop;
```

where the optional : T acts as a comment to the reader that the subtype of element E is T (and the compiler verifies this comment). *Use subtype_indication in generalized iterators (AI12-0156)* allows an optional subtype indication – though of the cursor not the element – to be given for the original **in** form of a **for** loop (see RM 5.5.2), i.e.:

```
for I : Index in Vec.Iterate loop
  Vec(I) := Vec(I) + 1;
end loop;
```

where Index is the subtype of the loop parameter.

4.4 Loop-body as anonymous procedure

Loop-body as anonymous procedure (AI12-0189) allows a loop body to be used to specify the implementation of a procedure to be passed as the actual for an access-to-subprogram parameter, when used in the context of a special kind of for loop statement, whose iterator specification is given by a procedure iterator (see RM 5.5.3).

This can be used for iterating over directories and environment variables, or iterating through a map-like container over the keys. Dedicated mechanisms were proposed for these (AI12-0009 and AI12-0188, respectively), but it was considered more useful to add a more general mechanism.

For example, to do something for each environment variable, in Ada 2012 the user can write a procedure and pass a pointer to it to the Iterate procedure of Ada.Environment_Variables, as in:

```
procedure Print_One
  (Name : in String;
   Value : in String) is
begin
   Put_Line (Name & "=" & Value);
end Print_One;
...
Ada.Environment_Variables.Iterate (Print_One'Access);
```

In Ada 2022, instead of explicitly writing a procedure, one can use the **of** form of a **for** loop, and the body of the loop is automatically turned into an anonymous procedure that is passed to the Iterate procedure of Ada.Environment_Variables, as in:

```
for (Name, Value) of Ada.Environment_Variables.Iterate (<>) loop
-- "(<>)" is optional because it is the last parameter
   Put_Line (Name & " => " & Value);
end loop;
```

An **exit**, **return**, **goto**, or other transfer of control out of the loop is only allowed if the iterator procedure has new aspect Allows_Exit with value True. Even if Allows_Exit is False, the loop can still end prematurely due to the propagation of an exception.

Allows_Exit aspect should be used on language-defined subprograms (AI12-0286) adds aspect Allows_Exit to the language-defined subprograms, where appropriate, i.e. to the Search procedure of Ada.Directories, and the Iterate procedure of Ada.Environment_Variables (see below, and used in the examples above and below).

```
procedure Iterate
    (Process : not null access procedure
      (Name, Value : in String))
    with Allows_Exit;
```

This allows an early exit when iterating over the environment variables, as in this more sophisticated variant of the previous example:

```
for (Name, Value) of Ada.Environment_Variables.Iterate (<>) loop
  if Bounded_Strings.Index
    (Source  => Bounded_Strings.To_Bounded_String (Name),
     Pattern => "Path",
     From    => Name'First) /= 0 then
      -- Found an environment variable with name "*Path*"
      Put_Line (Name & " => " & Value);
      exit;  --  Exit loop now
  end if;
end loop;
```

Bounded errors associated with procedural iterators (AI12-0326-2) extends AI12-0189 to make it a bounded error for an Allows_Exit subprogram to call the loop body procedure from an abort-deferred operation (unless the whole loop_statement was within this same abort-deferred operation), as this would interfere with implementing a transfer of control.

It is also adds the reserved word **parallel** to the syntax for procedural iterators, and makes it a bounded error to call a loop body procedure from multiple logical threads of control unless **parallel** is specified.

Note that if Allows_Exit is True then this precludes calling the loop body procedure from a parallel iterator.

It is anticipated that (internally) early exit will be implemented using exception handling. *Procedural iterator aspects (AI12-0344)* says that if restriction No_Exceptions is in force then an early exit will not be allowed, regardless of whether Allows_Exit is True.

If the reserved word **parallel** occurs in the loop statement for a loop body procedure then the Parallel_Calls aspect (see 2.2, “Nonblocking and data race checks”) is implicitly set for the loop body procedure.

4.5 Container aggregates

Currently, it is quite tedious to initialise a container, one has to create it as an empty container and then add elements one at a time, as in:

```
X : My_Set := Empty_Set;
Include (X, 1);
Include (X, 2);
Include (X, 3);
```

Container aggregates; generalized array aggregates (AI12-0212) adds positional container aggregates (see RM 4.3.5). These allow the above to be replaced by simply:

```
X : My_Set := [ 1, 2, 3 ];
```

Note that this uses square brackets not round brackets (parentheses). This allows the use of [ ] to indicate an empty container, analogous to "" indicating an empty string.

This is achieved using the new aspect Aggregate to indicate the appropriate function for returning an empty container of the particular container type, and also the appropriate procedure for adding an element to the particular container type.

For example:

```
type Set is tagged private
  with -- Ada 2012 has these
     Constant_Indexing => Constant_Reference,
     Default_Iterator  => Iterate,
     Iterator_Element  => Element_Type,
     ... -- but this is new
     Aggregate         => (Empty          => Empty,
                           Add_Unnamed => Include),
     ...
```

Originally an Empty_<Container> constant was asked for in AI12-0212, but *Empty function for Container aggregates (AI12-0339)* tweaked this such that now an Empty function is given instead, so as to allow a Capacity parameter for those container types that have the concept of capacity (e.g. Vectors).

Add_Unnamed requires a two parameter procedure for adding a single element. As the existing Append procedures of vectors and lists required a third, Count, parameter, *Contracts for container operations (AI12-0112)* added a procedure without the Count parameter (at the time called Append_One) to the Vectors container, and *List containers need Append_One (AI12-0391)* did the same for the Doubly_Link_Lists container. *Ambiguities associated with Vector Append and container aggregates (AI12-0400)* then decided it was cleaner for the new procedures to be overloadings of Append, and to remove the default for Count (of := 1) from the original Append procedures.

Iteration is also possible within the container aggregate, for example to create a set whose elements all have double the value of the corresponding elements of another set:

```
Doubles_Set : My_Set := [ for Item of X => Item * 2 ];
```

Note that this uses similar syntax to that introduced by *Index parameters in array aggregates (AI12-0061)* (see 7.4).

And – prepare yourself for a shock! – array aggregates are also allowed to use square brackets as an alternative to round brackets (parentheses). This is to emphasise the similarity in characteristics between containers and arrays, allow the use of [ ] for an empty array, and allow the use of positional notation for a single element array. Remember that

```
Two_Array : array (1 .. 2) of Positive := (1, 2);
```

is allowed, but not:

```
One_Array : array (1 .. 1) of Positive := (1); -- Illegal.
```

AI12-0212 also introduces two other kinds of container aggregates. These work similarly to the positional container aggregates already described, so we won't describe the details of defining these.

Named container aggregates allow the specification of key choices along with element values, just like a named array aggregate. For a named container aggregate, the choices can be of any type, so these aggregates are especially useful for maps, as they can give the keys and elements of a map at once. For instance, assuming an appropriate map has been defined, one could write:

```
Retired : Expert_Map := ["Jeff" => True, "Randy" => False, "Tuck" => False];
```

Indexed container aggregates are similar to named container aggregates except that they require the key choices to be of a discrete type. In exchange for this requirement, indexed container aggregates work similarly to an array aggregate, including a compile-time check that all of the specified choices cover a single range with no duplicates. Thus, using indexed container aggregates can help eliminate errors. For instance:

```
Part_Time : Employee_Vector := [1 | 3 | 5..10 => False, 2 | 4 => True]; -- OK.
On_Vacation : Employee_Vector := [1 .. 4 | 8 .. 10 => False, 5 | 7 => True]; -- Illegal.
```

The second aggregate is illegal as employee #6 is not specified.

One difference between container aggregates and array aggregates is that no **others** choice is allowed in container aggregates. An **others** choice only makes sense when the number of elements in the resulting object is known at compile-time, and that is never the case for containers.

Unfortunately introducing aggregates for containers introduced an ambiguity – for the Append, Insert and Prepend operations of vectors, if the element type is a record then it is unclear whether it is an element or another vector that is being added. *Ambiguities associated with Vector Append and container aggregates (AI12-0400)* renames the operations for adding a vector to Append_Vector, Insert_Vector and Prepend_Vector. Note that this is a backward incompatibility; compiler vendors are strongly encouraged to ease this incompatibility by adding these newly named routines to their Ada 2012 implementations.

4.6 Iterator filters

When iterating through a container, it is often required to filter the results to only return those values that meet some condition. *Iterator filters (AI12-0250)* adds filters to Ada 2022 to meet this need (see RM 5.5). This feature makes use of the keyword **when**, for example:

```
S : constant Set := (for E of C when E mod 2 = 1 => E);
```

to obtain all the odd elements of Container C.

While most useful in aggregate iterator associations, filters can be used anywhere an iterator is allowed in Ada, including quantified expressions, reduction expressions (see 2.5), and even loop statements. Similarly, iterator filters can be used with any kind of iterator, including the traditional discrete iterator. So a simple for loop like:

```
for I in 1 .. 100 loop
   if I mod 2 = 0 then
      -- Only process even values.
      ...
   end if;
end loop;
```

could instead be written as:

```
for I in 1 .. 100 when I mod 2 = 0 loop
   -- Only process even values.
   ...
end loop;
```

4.7 Iterators in array aggregates

In addition to container aggregates, *Container aggregates; generalized array aggregates (AI12-0212)* adds iterator associations to array aggregates (see RM 4.3.3), in order to keep the capabilities of container aggregates and array aggregates roughly the same. This feature allows the use of iterators to construct an array object directly without needing to write a loop statement.

If an array aggregate contains an iterator association then no other associations are allowed other than other iterator associations (there may be more than one of these). Since the number of items is only

known at runtime, the lower bound of the array value is determined as for a positional array aggregate, and the upper bound is determined by the number of actual values in the array value.

Array aggregate iterator associations can have filters (see 4.6) which allow the selection of specific elements to create an array. One could create an array of all of elements of a container C with all of the even elements first:

```
type Arr is array (Positive range <>) of Natural;
A : constant Arr :=
    [for E of C when E mod 2 = 0 => E,
     for E of C when E mod 2 = 1 => E];
```

Note that an iterator over a discrete range is provided by the ability to specify an index parameter (as introduced by *Index parameters in array aggregates (AI12-0061)* – see 7.4); unlike other iterator associations, it retains the usual compile-time aggregate checks and capabilities.

4.8 Indefinite Holders

Bounded_Indefinite_Holders (AI12-0254) adds a new container type, Bounded_Indefinite_Holder (see RM A.18.32), which allows the storage of a (single) class-wide object without the use of dynamic memory allocation, for use in safety critical environments. Rather than having a bounded indefinite variant of every container, it is envisaged that this holder container would be used as a building block, e.g. in a container of such holder containers.

Compared with the existing Indefinite_Holder, there is an additional generic parameter:

```
Max_Element_Size_in_Storage_Elements : Storage_Count;
```

If this is exceeded, Program_Error is raised.

Swap for Indefinite_Holders (AI12-0350) adds a Swap operation to both Indefinite_Holder and the new Bounded_Indefinite_Holder. For the former this avoids the overhead of copying the element (and any associated Adjust/Finalize).

```
procedure Swap (Left, Right : in out Holder)
  ...
```

4.9 Parallel Container Iterator filters

This topic, from *Parallel Container Iterator filters (AI12-0266)* is covered in the Parallelism section (see 2.1).

Chapter 5: Internationalisation

5.1 Additional internationalisation of Ada

Additional internationalisation of Ada (AI12-0021) adds child packages Wide_File_Names and Wide_Wide_File_Names for each I/O package (i.e. Sequential_IO, Direct_IO, Text_IO and Stream_IO), containing just those operations that take a filename as a parameter, and Wide_ and Wide_Wide_ versions of Ada.Directories, Ada.Command_Line and also of Ada.Environment_Variables.

Chapter 6: Real-Time

A number of AIs were discussed by, or arose from, the 19th International Real-Time Ada Workshop, as reported on in the Vol. 39, No. 2, March 2018 edition of the AUJ:

- *Thread-safe Ada libraries (AI12-0139-1).* This was subsequently dropped;
- *Deadline Floor Protocol (AI12-0230-1);*
- *Compare-and-swap for atomic objects (AI12-0234-1);*
- *Admission Policy Defined for Acquiring a Protected Object Resource (AI12-0276-1);*
- *Dispatching Needs More Dispatching Points (AI12-0279-1);*
- *CPU Affinity for Protected Objects (AI12-0281-1);*
- *Atomic and Volatile generic formal types (AI12-0282-1).*

IRTAW also proposed an extended version of the Ravenscar profile called Jorvik.

6.1 Specifying Nonblocking

This is covered in the Parallelism section (see 2.3) as the driving reason for adding it was safe parallelism, but it is more generally useful, for timing analysis or deadlock avoidance, for example, or simply documenting behaviour that some coding standards would ask for in a comment.

Blocking was already disallowed for protected types, but that wasn't detected so bounded errors were possible. Applying pragma Detect_Blocking causes a run-time check, but this has a performance overhead, whereas applying aspect Nonblocking to a protected type declaration causes a check at compile-time.

6.2 Exact size access to parts of composite atomic objects

Exact size access to parts of composite atomic objects (AI12-0128-1) specifies that memory accesses to subcomponents of an atomic composite object must read or write the entire object (see RM C.6). For example:

```
type Status is
  record
    Ready : Boolean;
    Length : Integer range 0 .. 15;
  end record;
for Status use
   record
    Ready at 0 range 0 .. 0;
    Length at 0 range 1 .. 5;
  end record;
Status_Register : Status
  with Address => ...,
       Size => 32,
       Atomic => True;
if Status_Register.Ready then -- Reads entire register
  null;
end if;
Status_Register.Length := 10; -- Prereads entire register, then writes entire register.
```

This is useful for controlling accesses to memory mapped device registers, which often require reads or writes to be to the entire register.

6.3 Max_Entry_Queue_Length aspect for entries

Max_Entry_Queue_Length aspect for entries (AI12-0164-1) defines the new aspect Max_Entry_Queue_Length for an entry declaration (see RM D.4). This specifies the maximum number of callers allowed on that entry. This facilitates timing analysis and should be useful for new restricted tasking profiles besides Ravenscar.

Violation of this restriction results in the raising of Program_Error at the point of the call or requeue.

The value specified for the Max_Entry_Queue_Length aspect for an entry must be no higher than any specified for the corresponding type, and both must be no higher than the Max_Entry_Queue_Length partition-wide restriction. These are checked at compilation.

6.4 Deadline Floor Protocol

Deadline Floor Protocol (AI12-0230-1) updates the EDF (Earliest Deadline First) policy (see RM D.2.6) in line with the latest thinking from the IRTAW Workshops. It now uses the Deadline Floor Protocol (DFP) in preference to the Stack Resource Protocol (SRP). This should not result in any backward compatibility problems as it is believed that currently no Ada implementations support EDF.

6.5 Atomic Operations

A family of atomic operations is added to optional Annex C Systems Programming by *Compare-and-swap for atomic objects (AI12-0234)*, *Support for Arithmetic Atomic Operations and Test and Set (AI12-0321)* and *Add a modular atomic arithmetic package (AI12-0364)*.

In systems where processors communicate via shared memory, there are two common methods of synchronisation. Thus "Compare-and-swap" atomically compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a given new value. "Test-and-set" modifies the contents of a memory location and returns its old value in a single atomic operation. These can then be used to construct spin locks. Such instructions are often available in the hardware. This AI makes them available in a more portable manner, assuming of course that the underlying hardware provides them. Note that which instructions can be performed atomically can vary even between processors in the same family.

These AIs add a number of library packages, namely

System.Atomic_Operations.Test_And_Set,

and the generic packages

System.Atomic_Operations.Exchange,
System.Atomic_Operations.Integer_Arithmetic,
System.Atomic_Operations.Modular_Arithmetic

The last three packages are generic so that they can be instantiated with an actual of the appropriate size for the memory architecture in use.

Compare-and-swap (see RM C.6.2):

```
generic
   type Atomic_Type is private with Atomic;
package System.Atomic_Operations.Exchange
      with Pure, Nonblocking is
   function Atomic_Exchange
         (Item  : aliased in out Atomic_Type;
          Value : Atomic_Type) return Atomic_Type
      with Convention => Intrinsic;
   function Atomic_Compare_And_Exchange
         (Item  : aliased in out Atomic_Type;
          Prior : aliased in out Atomic_Type;
          Desired : Atomic_Type) return Boolean
      with Convention => Intrinsic;
   function Is_Lock_Free (Item : aliased Atomic_Type)
         return Boolean
      with Convention => Intrinsic;
end System.Atomic_Operations.Exchange;
```

Atomic_Exchange atomically assigns the value of Value to Item, and returns the previous value of Item.

Atomic_Compare_And_Exchange first evaluates the value of Prior. It then performs the following steps as part of a single indivisible operation:

- evaluates the value of Item;
- compares the value of Item with the value of Prior;
- if equal, assigns Item the value of Desired;
- otherwise, makes no change to the value of Item.

After these steps, if the value of Item and Prior did not match, Prior is assigned the original value of Item, and the function returns False. Otherwise, Prior is unaffected and the function returns True.

Is_Lock_Free returns whether all the operations of the child package can be provided lock-free for a given object.

An example of a spin lock using Atomic_Exchange:

```
   type Atomic_Boolean is new Boolean with Atomic;
   package Exchange is
      new Atomic_Operations.Exchange
         (Atomic_Type => Atomic_Boolean);
   Lock : aliased Atomic_Boolean := False;

...

begin -- Some critical section, trying to get the lock:
   -- Acquire the lock
   while Exchange.Atomic_Exchange (Item => Lock, Value => True) loop
      null;
   end loop;
   ...                                -- Do stuff
   Lock := False;                     -- Release the lock
end;
```

For non-preemptive scheduling, it might be appropriate to call Ada.Dispatching.Yield rather than having a **null** statement.

Test-and-set (see RM C.6.3):

```
package System.Atomic_Operations.Test_And_Set
  with Pure, Nonblocking is
  type Test_And_Set_Flag is
      mod <implementation-defined>
    with Atomic, Default_Value => 0,
         Size => <Implementation-Defined>;
  function Atomic_Test_And_Set
      (Item : aliased in out Test_And_Set_Flag) return Boolean
    with Convention => Intrinsic;
  procedure Atomic_Clear
      (Item : aliased in out Test_And_Set_Flag)
    with Convention => Intrinsic;
  function Is_Lock_Free
      (Item : aliased Test_And_Set_Flag) return Boolean
    with Convention => Intrinsic;
end System.Atomic_Operations.Test_And_Set;
```

Atomic_Test_And_Set performs an atomic test-and-set operation on Item. Item is set to some implementation-defined non-zero value. The function returns True if the previous contents were non-zero, and otherwise returns False.

Atomic_Clear performs an atomic clear operation on Item. After the operation, Item contains zero.

An example of a spin lock using Atomic_Test_And_Set:

```
begin -- Some critical section, trying to get the lock:
  -- Acquire the lock
  while Atomic_Test_And_Set (Item => Lock) loop
    null;
  end loop;
  ... -- Do stuff
 Atomic_Clear (Item => Lock); -- Release the lock
end;
```

Atomic arithmetic (see RM C.6.4):

```
generic
   type Atomic_Type is range <> with Atomic;
package System.Atomic_Operations.Integer_Arithmetic
      with Pure, Nonblocking is
   procedure Atomic_Add
      (Item  : aliased in out Atomic_Type;
       Value : Atomic_Type)
    with Convention => Intrinsic;
   procedure Atomic_Subtract
      (Item  : aliased in out Atomic_Type;
       Value : Atomic_Type)
    with Convention => Intrinsic;
   function Atomic_Fetch_And_Add
     (Item  : aliased in out Atomic_Type;
      Value : Atomic_Type) return Atomic_Type
    with Convention => Intrinsic;
   function Atomic_Fetch_And_Subtract
      (Item  : aliased in out Atomic_Type;
       Value : Atomic_Type) return Atomic_Type
    with Convention => Intrinsic;
   function Is_Lock_Free (Item : aliased Atomic_Type)
               return Boolean
    with Convention => Intrinsic;
end System.Atomic_Operations.Arithmetic;
```

As one might expect, Atomic_Add and Atomic_Subtract atomically perform add and subtract, whereas Atomic_Fetch_And_Add and Atomic_Fetch_And_Subtract additionally return the original value of the Item.

Modular arithmetic (see RM C.6.5):

In this case, the package

System.Atomic_Operations.Modular_Arithmetic

has the same declaration as

System.Atomic_Operations.Integer_Arithmetic,

except that the formal parameter is:

```
type Atomic_Type is mod <> with Atomic;
```

6.6 Admission policy for protected objects

On multiprocessor systems, a spin lock is typically used to gain access to a protected object in order to execute a protected action. Most multiprocessor locking algorithms prescribe that if there is more than one request competing for the same resource, the requests are served in FIFO order. This bounds the time it takes for a lower priority task to gain access to a protected object. With Ravenscar all tasks are statically allocated to processors, and if each protected object used by tasks on different processors is given a high ceiling priority then the blocking time for each task can be computed.

Previously the language did not presume any ordering or queuing for tasks competing to start a protected action, *Admission policy defined for acquiring a protected object resource (AI12-0276)* adds:

```
pragma Admission_Policy (FIFO_Spinning);
```

to specify that FIFO_Spinning is used (see RM D.4.1). A task will inherit the ceiling priority of the protected object. Other, implementation-defined, Admission_Policies may also be specified.

6.7 Nonpreemptive dispatching needs more dispatching points

In non-preemptive dispatching there needs to be sufficient points where rescheduling can occur, so as to restrict the amount of time that a low priority task can block a higher priority task. If the low priority task "gets lucky" and the entries it calls are open, and suspension objects true, it can be some time before a rescheduling point occurs.

Nonpreemptive dispatching needs more dispatching points (AI12-0279) provides a solution to this is by defining a Yield aspect that can be specified for a subprogram (see RM D.2.1). If a reschedule has not occurred within a call of the subprogram, then one is inserted at the return from the subprogram.

6.8 CPU Affinity for Protected Objects

CPU Affinity for Protected Objects (AI12-0281) allows the CPU aspect to be applied to a protected type, not just a task type (see RM D.16). If all tasks that invoke protected operations of a protected object are on the same CPU as the protected object then it is possible for the runtime to avoid the overhead of acquiring a lock, and also avoid the risk of deadlock.

Program_Error is raised if a task on one CPU attempts to invoke a protected operation of a protected object on another CPU.

6.9 Atomic, Volatile, and Independent generic formal types

Atomic, Volatile, and Independent generic formal types (AI12-0282) is covered in 3.8, "Aspects for Generic Formal Parameters".

6.10 Jorvik Profile

The new *Jorvik Profile (AI12-0291)* is less restrictive than the Ravenscar profile (see RM D.13), but still allows timing and storage analyses. Most of the restrictions are the same, but restrictions

```
No_Implicit_Heap_Allocations,
No_Relative_Delay,
Max_Protected_Entries => 1,
No_Dependence => Ada.Calendar,
No_Dependence => Ada.Synchronous_Barriers
```

are omitted and the restriction Simple_Barriers is replaced by the weaker Pure_Barriers.

Restriction Pure_Barriers (AI12-0290) defines Pure_Barriers. Such barriers do not have to be simple Boolean local variables, but can be more complex Boolean expressions, as long as they do not have side effects, exceptions, or recursion. Additionally, the 'Count attribute is allowed in entry barriers, not just protected entry bodies.

Relaxing barrier restrictions (AI12-0369) allows the barrier expression to refer to subcomponents of the protected type even when subject to either of the restrictions Simple_Barriers or Pure_Barriers.

6.11 Fixes for Atomic and Volatile

Fixes for Atomic and Volatile (AI12-0363) contains the following improvements:

'Access may not be taken of a non-atomic subcomponent of an atomic object.

The nesting of atomic objects gives much scope for confusion, but to disallow them would be backwardly incompatible, so a new aspect Full_Access_Only is added. This can be applied to atomic and volatile types and objects to indicate that no atomic (or full access) objects are permitted as subcomponents. If any subcomponent of a full access object is accessed, then the whole object has to be accessed, by an atomic read followed by an atomic write.

An Atomic aspect of True now additionally indicates that Volatile and Independent are True.

Chapter 7: Others

The heading "Others" is not meant to imply that the features in this section are less important; indeed this section includes some of the most useful new features of Ada 2022. They are likely to be amongst the first to be implemented, and the first that programmers will want to use.

Some of the changes are to tidy up inconsistencies in the language, such as *Missing operations of static string types (AI12-0201)* and *Make objects more consistent (AI12-0226)*.

7.1 'Image for all types

It must be a shock to programmers coming from other languages, such as Python, that in Ada one couldn't directly output the value of a composite, but had to laboriously write one's own routine to do it field by field, then repeat if there was nesting of composites. Such a mechanistic process is more efficiently performed by a compiler than a programmer. And remember that, prior to *Add Object'Image (AI12-0124)*, included in the Ada 2012 Technical Corrigendum, there was also the tedium of having to look up the subtype for an object and use My_Subtype'Image (My_Object) to obtain the image of an object.

'Image for all types (AI12-0020) adds the attribute 'Image for all types (see RM 4.10). It should be a boon for debugging.

Following on from this, *Image attributes of language-defined types (AI12-0304)* requires that 'Image works for the language defined container types. This uses the new [ ] array aggregate syntax from *Container aggregates; generalized array aggregates (AI12-0212)*. For Maps it uses the form of a named array aggregate, e.g.:

```
[ Key1 => Value1, Key2 => Value2 ]
```

for Trees the form is a positional array aggregate, e.g.:

```
[ [ 1, 2 ] , [ 111, 222, 333 ] ]
```

for null containers the form is a null array aggregate, i.e.:

```
[ ]
```

Put_Image should use a Text_Buffer (AI12-0340) adds package Ada.Strings.Text_Buffers (see RM A.4.12), to make it easier for users to redefine 'Image for their own purposes.

7.2 The Fortran Annex needs updating to support Fortran 2008

The Fortran Annex needs updating to support Fortran 2008 (AI12-0058) updates the Fortran section of mandatory Annex B (see RM B.5) to support Fortran 2008, in particular better support for double precision complex arithmetic. Permissions corresponding to non-standard extensions, or implementation advice that is now considered to be bad practice, have been removed.

7.3 Object_Size attribute

Object_Size attribute (AI12-0059) adds the Object_Size attribute (see RM 13.3). Users have been after this since 1983! S'Object_Size denotes the size of an object of subtype S. It can be specified, but must be specified to a value that the compiler is able to allocate (usually an entire storage unit for most implementations).

S'Object_Size is an improvement on S'Size (which cannot be redefined without breaking existing code). Reading S'Size is not terribly useful as it just gives the theoretical minimum number of bits required for a

value of a given range, not the number of bits that the compiler is actually going to allocate to an object of the type. Specifying S'Size just gives a minimum, the compiler may allocate more. In contrast, S'Object_Size specifies the exact size that will be allocated in the absence of the specification of the representation in some other way (for instance, by specifying X'Size for an object X or by specifying the layout of a record component with a record representation clause).

7.4 Index parameters in array aggregates

Index parameters in array aggregates (AI12-0061) adds an optional index in array aggregates (see RM 4.3.3). Consider:

```
subtype Index is Positive range 1 .. 10;
type Array_Type is array (Index) of Positive;
Squares_Array : Array_Type := (for I in Index => I * I);
```

This provides a means of creating an aggregate when the element type is limited, provides a better means of initialising an array with a type invariant, and should be useful for everyday programming.

While these associations look like an iterator association in an array aggregate (see 4.7), they actually are closely related to normal named associations, with the same usage rules. Thus, multiple associations are allowed (so long as all of the choices are static), with or without index parameters in those other associations. Similarly, the usual completeness checks apply to an aggregate with these associations. Since these associations are not iterators, they do not allow iterator filters (see 4.6), as filters would make the completeness checks impossible.

7.5 Static expression functions

The aspect Static is introduced by *Static expression functions (AI12-0075)* (see RM 6.8). It can only be applied to an expression function, and requests that it be regarded as a static function.

When called in a context that requires a static expression, the actual parameters of the expression function need to be static. For example, if we declare:

```
function If_Then_Else (Flag : Boolean; X, Y : Integer) return Integer is
  (if Flag then X else Y) with Static;
```

and then attempt to declare:

```
V : Integer := 10;
X : constant := If_Then_Else (True, 37, V); -- Error.
```

we get an error at compile time since V is not a static expression.

Predefined shifts and rotates should be static (AI12-0385) applies this new aspect Static to the predefined shift and rotate functions of package Interfaces.

7.6 Aggregates and variant parts

Aggregates and variant parts (AI12-0086) allows a discriminant that controls a variant in an aggregate to be non-static if the subtype of the discriminant is static and all values belonging to that subtype select the same variant (see RM 4.3.1). For example:

```
type Enum is (Aa, Bb, Cc, ..., Zz);
subtype S is Enum range Dd .. Hh;
type Rec (D : Enum) is record
  case D is
    when S => Foo,
      Bar : Integer;
    when others =>
      null;
  end case;
end record;
function Make (D : S) return Rec is
begin
  return (D => D, Foo => 123, Bar => 456);
end;
```

7.7 @ as an abbreviation for the LHS of an assignment

The addition of the "target name symbol" by *Add @ as an abbreviation for the LHS of an assignment (AI12-0125-3)* (see RM 5.2.1) – the use of a single character placeholder for the left hand side of an assignment – has proved to be rather controversial with those who are used to Ada being verbose. However, one of the goals of Ada is to be readable, but having a lengthy name multiple times in the same statement and having to mentally determine whether occurrences were the same was not helping the readability of Ada code. In particular, the target name symbol can help make the intent of the code clearer. For instance, if you see:

```
My_Package.My_Array(I+1).Field :=
  My_Package.My_Array(I-1).Field + 1;
```

in Ada 2012, it's impossible to know if there is a "typo" here (with the I-1 intended to be I+1), or if something unusual is going on. Also, when in a hurry it would be very easy to assume that both of the names were the same (when they are not). If instead you see:

```
My_Package.My_Array(I+1).Field := @ + 1;
```

then the intent is clear, and there is much less chance of a typo.

This feature is similar in function to the += of the C family of languages. The Ada feature is more powerful though, being able to handle expressions such as series expansions. Here are a couple of examples:

```
My_Package.My_Array(I).Field :=
  My_Package.My_Array(I).Field ** 3 +
  My_Package.My_Array(I).Field ** 2 +
  My_Package.My_Array(I).Field;
```

could be shortened to:

```
My_Package.My_Array(I).Field := @ ** 3 + @ ** 2 + @;
```

and:

```
My_Package.My_Array(I).Field :=
  Natural'Min (My_Package.My_Array(I).Field, 1000);
```

could be shortened to:

```
My_Package.My_Array(I).Field := Natural'Min (@, 1000);
```

See 3.4 for another example of this kind.

7.8 Aggregates of Unchecked_Unions using named notation

Given that it is generally regarded as good practice to use named notation rather than positional notation, it was somewhat bizarre that Unchecked_Unions only allowed the latter. Both are now allowed by *Aggregates of Unchecked_Unions using named notation (AI12-0174)* (see RM B.3.3). For example:

```
type Data_Kind is (C_int, C_char);
type C_Variant (Format : Data_Kind := C_int) is record
    case Format is
      when C_int =>
        int_Val : C.int;
      when C_char =>
        char_Val : C.char;
    end case;
end record
  with Unchecked_Union, Convention => C;
Int1 : C_Variant := (C_int, 12); -- Always OK
Int2 : C_Variant := (Format => C_int, int_Val => 12);
            -- Was illegal, now OK
```

7.9 Preelaborable packages with address clauses

Preelaborable packages with address clauses (AI12-0175) allows packages with aspect Preelaborate to contain certain simple functions known to the compiler, i.e. an instance of Unchecked_Conversion, a function declared in System.Storage_Elements, or the functions To_Pointer and To_Address declared in an instance of System.Address_to_Access_Conversions. This allows the declaring of objects with an address clause within a preelaborable package, which can be very useful for small embedded systems.

7.10 Missing operations of static string types

Missing operations of static string types (AI12-0201) defines that relational operators and type conversions of static string types are now static.

Static membership tests for strings, e.g. S **in** "abc", were already allowed; static equality tests for strings, e.g. S = "abc", are now also allowed.

7.11 Big Numbers

Predefined Big numbers support (AI12-0208) defines a package Ada.Numerics.Big_Numbers, with child packages Big_Integers and Big_Reals, to support arbitrary precision arithmetic (see RM A.5.6 and A.5.7). Types Big_Integer and Big_Real, and the usual comparison and arithmetic operators, are provided in their respective child packages.

Changes to Big_Integer and Big_Real (AI12-0366) then updates AI12-0208 in the light of implementation experience.

Now for an example, using the RSA algorithm. Value *v* can be encrypted using:

$c = v^e \bmod n$

and decrypted using:

$v = c^d \bmod n$

Both encryption and decryption require a value to be raised to a power (which is often but not always a prime number) and then take the remainder when divided by n which is the product of two (typically large) primes. The fine details of how to choose e and d need not bother us. Executing:

```
function Power_Mod (M, Exp, N : Integer) return Integer is
    ((M ** Exp) mod N);
```

would soon overflow for all but the smallest of values. The risk of overflow can be reduced by taking the remainder for intermediate results, as in the following simple algorithm:

```
function Power_Mod (M, Exp, N : Integer) return Integer is
  Result : Integer := 1;
begin
  for Count in 1 .. Exp loop
    Result := (@ * M) mod N;
  end loop;
  return Result;
end Power_Mod;
```

This avoids overflow (and performs at least as well as more sophisticated algorithms) for the sort of values typically used in textbook examples. But real world encryption may use values hundreds of digits long, so Integer should be replaced by Big_Integer. An example using Big_Integer in a more sophisticated algorithm is:

```
with Ada.Numerics.Big_Numbers.Big_Integers;
use  Ada.Numerics.Big_Numbers.Big_Integers;
function Power_Mod (M, Exp, N : Big_Integer) return Big_Integer is
  Result      : Big_Integer := 1;
  Temp_Exp    : Big_Integer := Exp;
  Mult        : Big_Integer := M mod N;
begin
  while Temp_Exp > 1 loop
    if Temp_Exp mod 2 /= 0 then
      Result := (@ * Mult) mod N;
    end if;
    Mult := @ ** 2 mod N;
    Temp_Exp := @ / 2;
  end loop;
  Result := (@ * Mult) mod N;
  return Result;
end Power_Mod;
```

In order to make Big Numbers easier to use, *User-defined numeric literals (AI12-0249)* allows the user to define numeric literals to be used with a (non-numeric) type, using aspects Integer_Literal and Real_Literal to identify a function that will do the interpretation (see RM 4.2.1). For example:

```
type Big_Integer is private
  with Integer_Literal => Big_Integer_Value;
function Big_Integer_Value (S : String) return Big_Integer;
...
Y : Big_Integer := -3;
```

This is equivalent to:

```
Y : Big_Integer := - Big_Integer_Value ("3");
```

Named Numbers and User-Defined Numeric Literals (AI12-0394) extends this to allow named numbers to be used with user-defined numeric literals.

User-defined string literals (AI12-0295) allows the user to define string literals to be used with a non-string type, using aspect String_Literal to identify a function that will do the interpretation. For example:

```
type Varying_String is private
  with String_Literal => To_Varying_String;
function To_Varying_String (Source : Wide_Wide_String)
  return Varying_String;
...
X : constant Varying_String := "This is a test";
```

This is equivalent to:

```
X : constant Varying_String :=
     To_Varying_String (Wide_Wide_String'("This is a test"));
```

Various issues with user-defined literals (AI12-0325) and *Various issues with user-defined literals (part 2) (AI12-0342)* sort out some of the details of both user-defined numeric literals and user-defined string literals.

7.12 Objects, Values, Conversions and Renaming

There are several AIs that aim to make Ada more consistent in its treatment of type conversions and qualified expressions, objects and values.

Make objects more consistent (AI12-0226).

In Ada 2012 a type conversion between two tagged types (termed a view conversion) gives an object, which can be renamed, whereas a type conversion between two untagged types (termed a value conversion) gives a value, which cannot. In Ada 2022 a value conversion of an object is added to the list of things deemed to be an object, making it consistent with a qualified expression. If we have

```
Max : constant Natural := 10;
```

then the following are now all legal:

```
Ren1 : Natural renames Max;          -- Legal
Ren2 : Natural renames Natural'(Max); -- Qualified expression, legal
Ren3 : Natural renames Natural(Max); -- Value conversion, was illegal, now legal
```

Renaming values (AI12-0383). takes this further by allowing values to be renamed anyway, besides objects.

Make subtype_mark optional in object renames (AI12-0275).

This makes the subtype optional in object renaming declarations (see RM 8.5.1). The expression will be correctly typed as long as the right hand object can be resolved to only one specific type.

It has long been an irritant that writing a renaming declaration required looking up the subtype of the object, and giving the subtype can be misleading anyway. The reader may be surprised to find that the following is valid Ada:

```
   subtype My_Subtype         is Integer range 3 .. 5;
   subtype Garbage_Subtype is Integer range -19 .. -7;
   X : My_Subtype;
   Y : Garbage_Subtype renames X;
begin
   case Y is
      when 3 .. 5 =>
         pragma Assert (Y in 3 .. 5);
   end case;
```

The subtype given in the renaming merely has to be a subtype of the same type as the object being renamed. The constraints, null exclusions, and predicates of the subtype given in the renaming are ignored; instead they are taken from the subtype of the object being renamed. Note that the case statement has to have alternatives for each possible value of the subtype of X, not of the supposed subtype of Y.

Requiring the subtypes to match has been considered in the past, but backward compatibility concerns have always prevented this. For backward compatibility reasons the subtype may still be given, but it is (usually) no longer required.

Renaming of a qualified expression of a variable (AI12-0401).

Ada 2012 allows a qualified expression to be renamed (though possibly this had not been implemented by anyone until recently). In the following, three subtypes are involved:

```
   subtype Subtype_1 is Integer range ...;
   subtype Subtype_2 is Integer range ...;
   subtype Subtype_3 is Integer range ...;
   X : Subtype_1;
   Y : Subtype_2 renames Subtype_3'(X);
```

As described under *Make subtype_mark optional in object renames (AI12-0275)*, Subtype_2 is largely irrelevant. Typically, a qualified expression is used to confirm which type is meant when there is a possible ambiguity, as in:

```
type Traffic_Light is (Red, Amber, Green);
type Gemstones is (Amber, Ruby, Topaz);
... Traffic_Lights'(Amber) ...
... Gemstones'(Amber) ...
```

But in the above it is not a case of X being renamed, and Subtype_3' just being there to confirm the type; instead it is the whole qualified expression Subtype_3'(X) that is the object being renamed. Thus Y takes its constraints, null exclusions, and predicates from Subtype_3. This could be problematic if Subtype_3 has a narrower range than Subtype_1, as in the following:

```
   X : Natural := ...;
   Y : Positive renames Positive'(X); -- Legal in Ada 2012, illegal in Ada 2022
begin
   X := 0; -- Alert!!
   case Y is
      when 1 .. Positive'Last =>
         pragma Assert (Y > 0);
   end case;
```

X is set to a value outside of the range of Y, effectively bypassing the range check at the time of renaming, which is meant to be a one-off affair, and not have to be repeated at each subsequent use of Y.

To fix this hole, *Renaming of a qualified expression of a variable (AI12-0401)* requires Subtype_3 to match either Subtype_1 or its base type (in this case, Integer). Note that the converse problem of writing

a value to Y outside of the range of X does not arise as a qualified expression gives a constant (i.e. read-only) view.

7.13 Getting the representation of an enumeration value

It has long been possible, with Ada, to define non-default numeric values for an enumeration type, using a representation clause, as in:

```
type Status is (Off, Ready, On);
for Status use (Off => 1, Ready => 2, On => 4);
```

for interfacing to hardware where one, and only one, of a group of three bits should be set. It was not possible to subsequently query what numeric value represented a given enumeration literal, or vice versa, though. This may have been to discourage users from adopting a C mentality of thinking in terms of the numeric values. But to save the user from workarounds, typically involving Unchecked_Conversion, Ada 2022 (via *Getting the representation of an enumeration value AI12-0237)* provides attribute 'Enum_Rep to return the number representing a given enumeration literal, and 'Enum_Val to return the enumeration literal represented by a given number (see RM 13.4). Thus:

```
   My_Status    : Status := ...;
   Rep          : Integer;
begin
   Rep          := Status'Enum_Rep (My_Status);
   My_Status    := Status'Enum_Val (Rep);
```

7.14 Attributes for fixed point types

AI12-0362 says that an implementation is permitted to support Floor, Ceiling, and rounding attributes for fixed point types. The AI came in rather late in the day to mandate support, but there is sufficient support to add something.

Chapter 8: Conclusion

Ada 2022 will bring Ada into a new decade. May many more projects be delivered successfully using it!

Index

Entries in this index reference section numbers and not page numbers.

V

W

Y