An Invitation to Ada 2005

Pascal Leroy
Senior Software Engineer
Chairman, Ada Rapporteur Group

Rational software

Agenda

- Object-Oriented Programming
- Access Types
- Real-Time, Safety and Criticality
- General-Purpose Capabilities
- Predefined Environment and Interfacing

Object-Oriented Programming

- Preserve Ada's strengths for the construction of safe systems
  - Distinction between specific and class-wide types
  - Static binding by default, dynamic binding only when necessary
  - Strong boundary around modules
- Enhance object-oriented features
  - Multi-package cyclic type structures
  - Multiple-inheritance type hierarchies
  - Prefix notation
  - Accidental overloading or overriding
  - Extension for protected and task types

Multi-Package Cyclic Type Structures

- Impossible to declare cyclic type structures across library package boundaries
  - Problem existed in Ada 83
  - More prominent with the introduction of child units and tagged types
  - Workarounds result in cumbersome code

Multi-Package Cyclic Type Structures: Example

```
with Employees;
package Departments is
   type Department is tagged private;
   procedure Choose_Manager (D : in out Department;
                             Manager : in out Employees.Employee);
private
   type Emp_Ptr is access all Employees.Employee;
   type Department is tagged record
         Manager : Emp_Ptr;
      end record;
end Departments;
```

Illegal circularity!

```
with Departments;
package Employees is
   type Employee is tagged private;
   procedure Assign_Employee (E : in out Employee;
                              D : in out Departments.Department);
private
   type Dept_Ptr is access all Departments.Department;
   type Employee is tagged record
         Department : Dept_Ptr;
      end record;
end Employees;
```

Limited With Clauses

- Gives visibility on a *limited view* of a package
  - Contains only types and nested packages
  - Types behave as if they were incomplete
  - Restrictions on the possible usages of a limited view (no use, no renaming, etc.)
  - Cycles are permitted among limited with clauses
  - Imply some kind of “peeking” before compiling a package
- Related change: incomplete tagged types
  - Can be used as a parameter
  - Always passed by reference
  - Support for cycles in object-oriented programming

Limited With Clauses (cont'd)

```
package Departments is
   type Department is tagged;
end Departments;
```

limited view: implicit, visible through limited with

```
limited with Departments;
package Employees is
   type Employee is tagged private;
   procedure Assign_Employee (E : in out Employee;
                              D : in out Departments.Department);
private
   type Dept_Ptr is access all Departments.Department;
   type Employee is tagged record
         Department : Dept_Ptr;
      end record;
end Employees;
```

```
with Employees;
package Departments is
   type Department is tagged private;
   procedure Choose_Manager (D : in out Department;
                             Manager : in out Employees.Employee);
private
   type Emp_Ptr is access all Employees.Employee;
   type Department is tagged record
         Manager : Emp_Ptr;
      end record;
end Departments;
```

Multiple-Inheritance Type Hierarchies

- Multiple inheritance too heavy for Ada 95
- Java and C# have a lightweight multiple inheritance mechanism: interfaces
  - Relatively inexpensive at execution time
  - No conflicts due to inheriting code from multiple parents
- Add interfaces, similar to abstract types but with multiple inheritance
  - May be used as a secondary parent in type derivations
  - Have class-wide types
  - Support for composition of interfaces
  - No components, no objects
- Related change: null procedures
  - A procedure declared null need not be overridden

Interfaces: Example

```
type Model is interface;
type Model_Ref is access all Model'Class;

type Observer is interface;
procedure Notify
           (O : access Observer;
            M : Model_Ref) is abstract;

type View is interface with Observer;
type View_Ref is access all View'Class;
procedure Display
           (V : access View;
            M : Model_Ref) is abstract;

type Controller is interface with Observer;
type Controller_Ref is
   access all Controller'Class;
procedure Start
           (C : access Controller;
            M : Model_Ref) is abstract;

procedure Register
           (M : access Model;
            V : View_Ref) is abstract;
procedure Register
           (M : access Model;
            C : Controller_Ref) is abstract;
```

- Model-View-Controller Structure
  - Model: data structure being viewed and manipulated
  - Observer: waits for a change to a model
  - View: visual display of a model
  - Controller: supports input devices for a model
- Note composition of interfaces

Interfaces: Example (cont'd)

```
type Device is tagged private;
procedure Input (D : in out Device);

type Mouse is
   new Device and Controller with private;
procedure Input (D : in out Mouse);

procedure Start (D : access Mouse;
                 M : Model_Ref);
procedure Notify (D : access Mouse;
                  M : Model_Ref);

type Two_Button_Mouse is
   new Mouse with private;
procedure Start
            (D : access Two_Button_Mouse;
             M : Model_Ref);

procedure Register_And_Start
              (D : access Mouse'Class;
               M : Model_Ref);
```

- Mouse is a concrete type implementing interface Controller
  - Only one concrete parent, Device
  - Any number of interface parents
- Mouse inherits operations from all of its parents
  - May (but need not) override Input
  - *Must* override Start and Notify
- Two_Button_Mouse inherits all the operations of Mouse
  - May (but need not) override some of them
- Register_And_Start is a class-wide operation

Prefix Notation

- A call must identify the package in which an operation is declared
  - Dispatching operations are often implicitly declared
- Class-wide operations not inherited
  - Declared in the original package where they appear
- Hard to identify the package where an operation is declared
  - Difficulty compounded by the fact that the choice between dispatching and class-wide may be an implementation detail
  - Use clauses are unappealing

Prefix Notation (cont'd)

- Add support for the Object.Operation notation common in other object-oriented languages
  - Only for tagged types and access designating tagged types
  - Dispatching operations and class-wide operations declared in the same package as the type are eligible
  - First parameter of the subprogram must be a controlling parameter
  - Prefix passed as first parameter

```
M : Model_Ref;
V : View_Ref;
C : Controller_Ref;
D : aliased Mouse;
…
V.Display (M);              -- equivalent to Display (V, M)
D.Start (M);                -- equivalent to Start (D, M)
D.Input;                    -- equivalent to Input (D)
D.Register_And_Start (M);   -- equivalent to
                            --    Register_And_Start (D'Access, M);
```

Accidental Overloading or Overriding

- A typographic error may change overriding into overloading or vice-versa
- Optional syntax to specify that a subprogram is an override or an overload
  - For compatibility, the absence of a qualifier means “don’t know”

```
type Root_Type is new Ada.Finalization.Controlled with …;

overriding
procedure Finalize (Object : in out Root_Type); -- OK.


type Derived_Type is new Root_Type with …;

overriding
procedure Finalise (Object : in out Derived_Type); -- Error here.

not overriding
procedure Do_Something (Object : in out Derived_Type); -- OK.
```

Extensions for Protected and Task Types

- Type extensions might be useful for protected and task types in addition to record types
- Inheriting of code is complex, notably because of the difficulty to specify how guards and barriers are inherited
- Simpler approach: define interfaces for protected and task types
  - Includes support for composition of interfaces
  - A protected or task type may implement any number of interfaces
- Proposal still in a state of flux

Agenda

- Object-Oriented Programming
- Access Types
- Real-Time, Safety and Criticality
- General-Purpose Capabilities
- Predefined Environment and Interfacing

Generalized Use of Anonymous Access Types

- Most OO languages allow free conversion of a reference to a subclass to a reference to its superclass
  - Ada requires explicit conversions which degrade readability
- Allow anonymous access types in all contexts
  - Avoids most explicit conversions
  - Avoids proliferation of access types
  - Unsure about function returning anonymous access types, yet

Generalized Use of Anonymous Access Types: Example

```
type Animal is tagged …;
type Horse is new Animal with …;
type Pig is new Animal with …;

type Acc_Horse is access all Horse'Class;
type Acc_Pig is access all Pig;

Napoleon, Snowball : Acc_Pig := …;
Boxer, Clover : Acc_Horse := …;
Animal_Farm : constant array (Positive range <>) of
              access Animal'Class :=
                    (Napoleon, Snowball, Boxer, Clover);

type Noah_S_Arch is
   record
      Stallion, Mare : access Horse;
      Boar, Sow      : access Pig;
   end record;
```

Downward Closures for Access to Subprogram Types

- Access-to-subprogram types subject to accessibility checks
  - Necessary to prevent dangling references
  - Requires awkward idioms to deal with nested subprograms

```
type Integrand is access function (X : Float) return Float;

function Integrate (Fn : Integrand; Lo, Hi : Float) return Float;
```

- Anonymous access-to-subprogram types
  - Cannot be assigned
  - Cannot be used to create dangling references

```
function Integrate (Fn : access function (X : Float) return Float;
                    Lo, Hi : Float) return Float;
```

Constancy and Null Exclusion

- No access-to-constant parameters or discriminants in Ada 95
- Would be useful for:
  - Declaring controlling parameters of an operation that doesn't modify the designated object
  - Providing read-only access via a discriminant
  - Interfacing with other languages
- Literal null disallowed for anonymous access types
  - Causes confusion
  - Problematic when interfacing with a foreign language

Constancy and Null Exclusion (cont'd)

- Define an explicit way to exclude nulls from an access subtype
  - Make existing anonymous access types include null by default
- Provide a mechanism for declaring constant anonymous access types

```
type Non_Null_Ptr is not null access T;

-- X guaranteed to not be null.
procedure Show (X : Non_Null_Ptr);

-- Pass by reference, but don't allow designated object to be updated;
-- guarantee Y is non-null.
procedure Pass_By_Ref (Y : not null access constant Rec);

-- Any pointer to a graph may be passed to the display routine,
including null.
procedure Display (W : access Window;
                   G : access constant Graph'Class);
```

Agenda

- Object-Oriented Programming
- Access Types
- Real-Time, Safety and Criticality
- General-Purpose Capabilities
- Predefined Environment and Interfacing

Ravenscar Profile for High-Integrity Systems

- *De facto* standard defined by the IRTAW
  - Intended for use in high-integrity system
  - Makes it possible to use a reduced, reliable run-time kernel
  - Many capabilities generally useful for other application domains
- Add new restrictions and pragmas Detect_Blocking and Profile
- Define Ravenscar in terms of predefined restrictions and pragmas
  - Current users of Ravenscar virtually unaffected
  - Some application domains only need to abide by some of the restrictions, not the whole profile
  - Implementers may define new profiles for specific needs

Dynamic Ceiling Priorities

- Tasks have dynamic priorities in Ada 95
- Protected objects only have static ceiling priorities
  - Unfortunate for some applications
- Add attribute Priority
  - Prefix is a protected object
  - Gives the ceiling priority of the object
  - Attribute is a variable: may be updated, providing dynamic behavior
  - Completes the language in terms of dynamic priorities

Timing Events

- Some scheduling schemes require to execute code at a particular future time
  - To asynchronously change the priority of a task
  - To allow tasks to come off the delay queue at a different priority
- High priority “minder” task needed in Ada 95
  - Inefficient and inelegant
- Add a mechanism to allow user-defined procedures to be executed at a specified time
  - Without the need to use a task or a delay statement
- Provided by new predefined unit Ada.Real_Time.Timing_Events
  - Limited private type Timing_Event represents an event occurring at some time
  - Time may be absolute or relative
  - Protected procedure may be used to handle a timing event

Execution-Time Clocks and Budgeting

- Measuring execution time is fundamental for the safe execution of real-time systems
- Use of aperiodic servers to control allocation is becoming common; requires budget control
- New predefined package Ada.Real_Time.Execution_Time
  - Private type CPU_Time represents the CPU time consumed by a task
  - Handler called when a task has consumed a predetermined amount of CPU
  - Supports CPU-based scheduling
- New predefined package Ada.Real_Time.Execution_Time.Group_Budgets
  - Private type Group_Budget represents a CPU budget for use by a group of tasks
  - Operations to add or remove a task to a group
  - Operations to query the remaining budget and change the budget
  - Handler called when a budget has expired

Scheduling Mechanisms

- Ada 95 only has FIFO scheduling
  - Other policies may be defined by an implementation, but they are not portable
- Other scheduling techniques are used in practice
  - Round robin
  - Earliest deadline first
- Round robin is very common and fits well with the current FIFO
- Earliest deadline first is the preferred scheduling mechanism for soft real-time
  - Much better CPU usage (40% more before deadlines are missed)
- Add a mechanism to mix scheduling techniques in an application

Agenda

- Object-Oriented Programming
- Access Types
- Real-Time, Safety and Criticality
- General-Purpose Capabilities
- Predefined Environment and Interfacing

Access to Private Units in the Private Part

- Impossible to reference a private unit in the private part of a public package

- Private with clause gives visibility at the beginning of the private part

```
private package Claw.Low_Level_Image_Lists is
   …
end;

private with Claw.Low_Level_Image_Lists;
package Claw.Image_List is
   … -- May not use Low_Level_Image_Lists here.
private
   … -- May use Low_Level_Image_Lists here.
end;
```

Aggregates for Limited Types

- Limited types prevent copying of values
  - Have limitations unrelated to copying
  - Aggregates not available: no full coverage checking
- Allow aggregates for limited types
  - New syntax to force default initialization of some components

```
private protected type Semaphore is …;
type Object is limited
     record
          Sem: Semaphore;
          Size : Natural;
     end record;
type Ptr is access Object;

X : Ptr := new Object'(Sem => <>, Size => 0); -- Coverage checking.
```

Pragma Unsuppress

- Some algorithms may depend on the presence of canonical checks
  - Interactions with pragma Suppress may lead to bugs

- Pragma Unsuppress revokes the permission granted by Suppress

```
function "*" (Left, Right : Saturation_Type) return Saturation_Type is
     pragma Unsuppress (Overflow_Check);
begin
     return Integer (Left) * Integer (Right);
exception
     when Constraint_Error =>
          if (Left > 0 and Right > 0) or (Left < 0 and Right < 0) then
               return Saturation_Type'Last;
          else
               return Saturation_Type'First;
          end if;
end "*";
```

Agenda

- Object-Oriented Programming
- Access Types
- Real-Time, Safety and Criticality
- General-Purpose Capabilities
- Predefined Environment and Interfacing

Unchecked Unions: Variant Records with no Run-Time Discriminant

- No support in Ada 95 for interfacing with C unions
  - Unchecked_Conversion not satisfactory
- Pragma Unchecked_Union prevents discriminants from being stored
  - Operations that need to read a discriminant are either illegal or raise Program_Error

```
union {
   spvalue double;
   struct {
      length int;
      first  *double;
   } mpvalue;
} number;
```

```
type Number (Kind : Precision) is
   record
      case Kind is
         when Single_Precision =>
            SPValue : Long_Float;
         when Multiple_Precision =>
            MP_Value_Length : Integer;
            MP_Value_First : Access_Long_Float;
      end case;
   end record;
pragma Unchecked_Union (Number);
```

Vector and Matrix Operations

- ISO/IEC 13813 defined real and complex vectors and matrices for Ada 83
  - No support for basic linear algebra
  - Not provided by vendors
- Integrate this capability in Annex G (Numerics)
  - Two new predefined units: Ada.Numerics.Generic_Real_Arrays and Ada.Numerics.Generic_Complex_Arrays
  - Adapted for Ada 05
  - Add support for basic linear algebra: inversion, resolution, eigensystem
  - May be used as an interface to existing linear algebra libraries or as a self-standing implementation

Container Library

- Numerous container libraries available as public domain software
  - Stacks, lists, maps, sets, trees, queues, graphs, etc…
- Language-defined containers would improve portability and usability of the language
- Delegated by the ARG to outside groups
  - Not enough resources in the ARG to fully specify a bulky API
  - Proposals will be evaluated by the ARG

Directory Operations

- Modern operating systems have a tree-structured file system
- Applications need to manage these file systems
- New predefined package Ada.Directory_Operations
  - Query and set the current directory
  - Create and remove directories or directory trees
  - Copy and rename files and directories
  - Decompose and compose file and directory paths
  - Check the existence, size and modification time of a file
  - Iterate over files and directories

Conclusion

- Snapshot of work in progress
  - Other features are being considered
  - More work needed to integrate all the changes together: consistency, orthogonality
- Schedule-driven: expect completion around the end of 2005
  - Features frozen by the end of this year
  - Implementers may want to do pilot implementation of some new features, based on user demand
- Make Ada safer, more powerful, more appealing to new and existing users