Ada 9x Tagged Types

and their Implementation

in GNAT

Cyrille Comar & Brett Porter

Gnat Project

OUTLINE

1. Ada 9x Object-Oriented Concepts Through An Example

- The main GNAT Data-Structures with Tagged Types
- Extending and Dispatching

2. Dealing With Common User Mistakes

- Tagged types outside Package Specs.
- Freezing point Issues

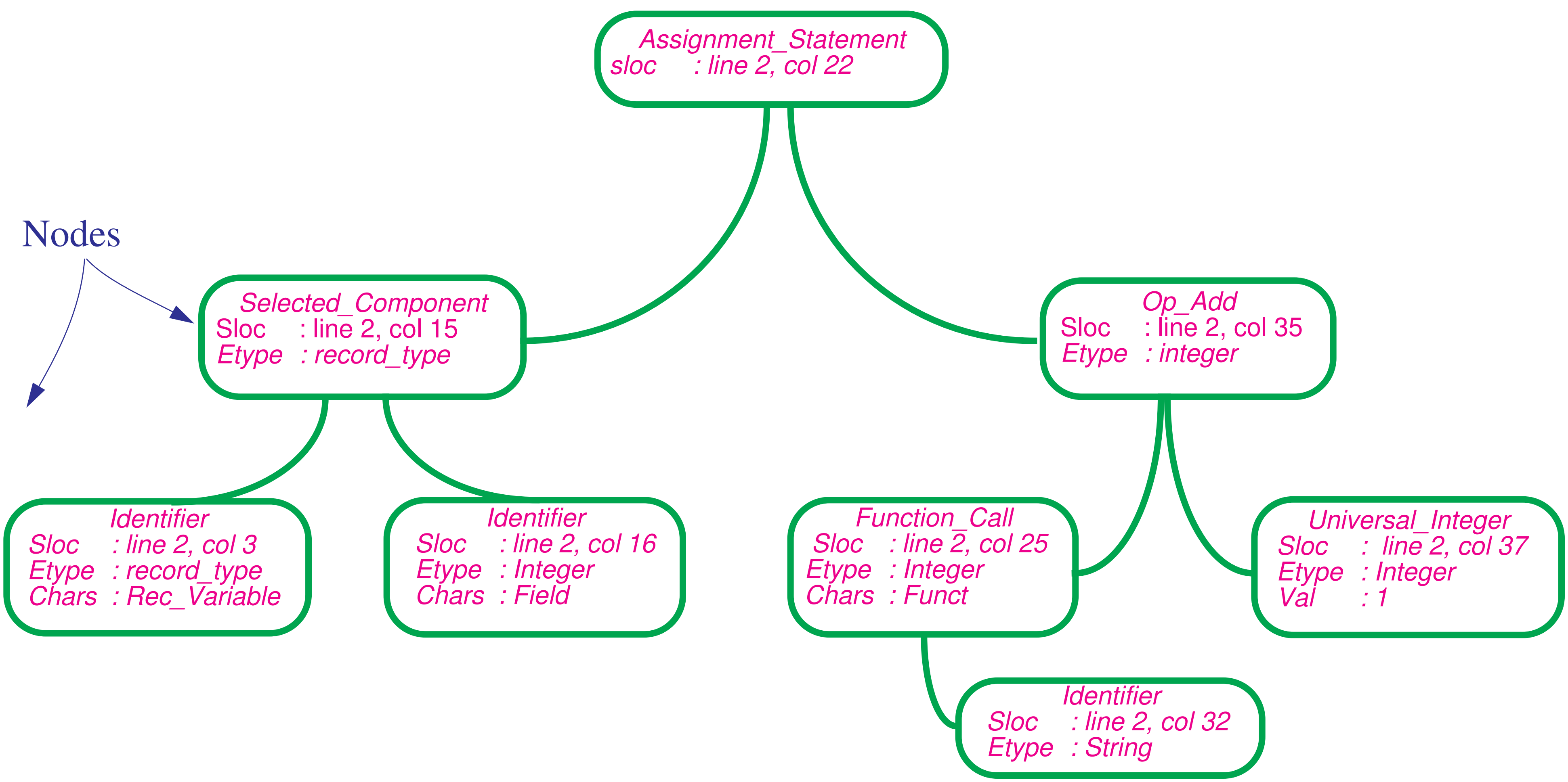
3. Basic Implementation Techniques

- Layout of Tagged Objects
- Structure of the Dispatch Table

4. The Low-Level C++ Interface

- Goals and Limitations
- The Gnat-specific Pragmas supporting the interface

a GNAT tree for `Rec_Variable.Field := Funct (X) + 1;`



Basic Syntax and Concepts

abstract tagged type

```
package Definitions is

   type Node is abstract tagged
      record
         Sloc : Source_Ptr;
      end record;
```

The corresponding classwide type

```
   type Node_Id is access Node'Class;
```

polymorphic pointer

```
   procedure Analyze_Node (N : Node_Id);
```

classwide operation

```
   procedure Analyze (N : access Node) is abstract;
   procedure Expand  (N : access Node) is abstract;
```

primitive operations

concrete type extension

```
type Assignment_Statement is new Node with
   record
      Name       : Expr_Id;
      Expression : Expr_Id;
   end record;
```

```
procedure Analyze (N : access Assignment_Statement);
procedure Expand  (N : access Assignment_Statement);
```

Overriding of primitive operations

class of an incomplete type

```
type Simple_Type;
type Type_Id is access Simple_Type'Class;

type Expr is abstract new Node with
   record
       Etype : Type_Id;
   end record;
```

abstract type extension

```
-- Inherited:
--  procedure Analyze (N : access Expr) is abstract;
--  procedure Expand  (N : access Expr) is abstract;
```

inheritance

```
procedure Resolve_Expr (E : Expr_Id; Typ : Type_Id);
procedure Resolve  (E : access Expr; Typ : Type_Id)
  is abstract;
```

Primitive Operations versus Operations on Classwide Types

This is a general algorithm, valid for all Nodes

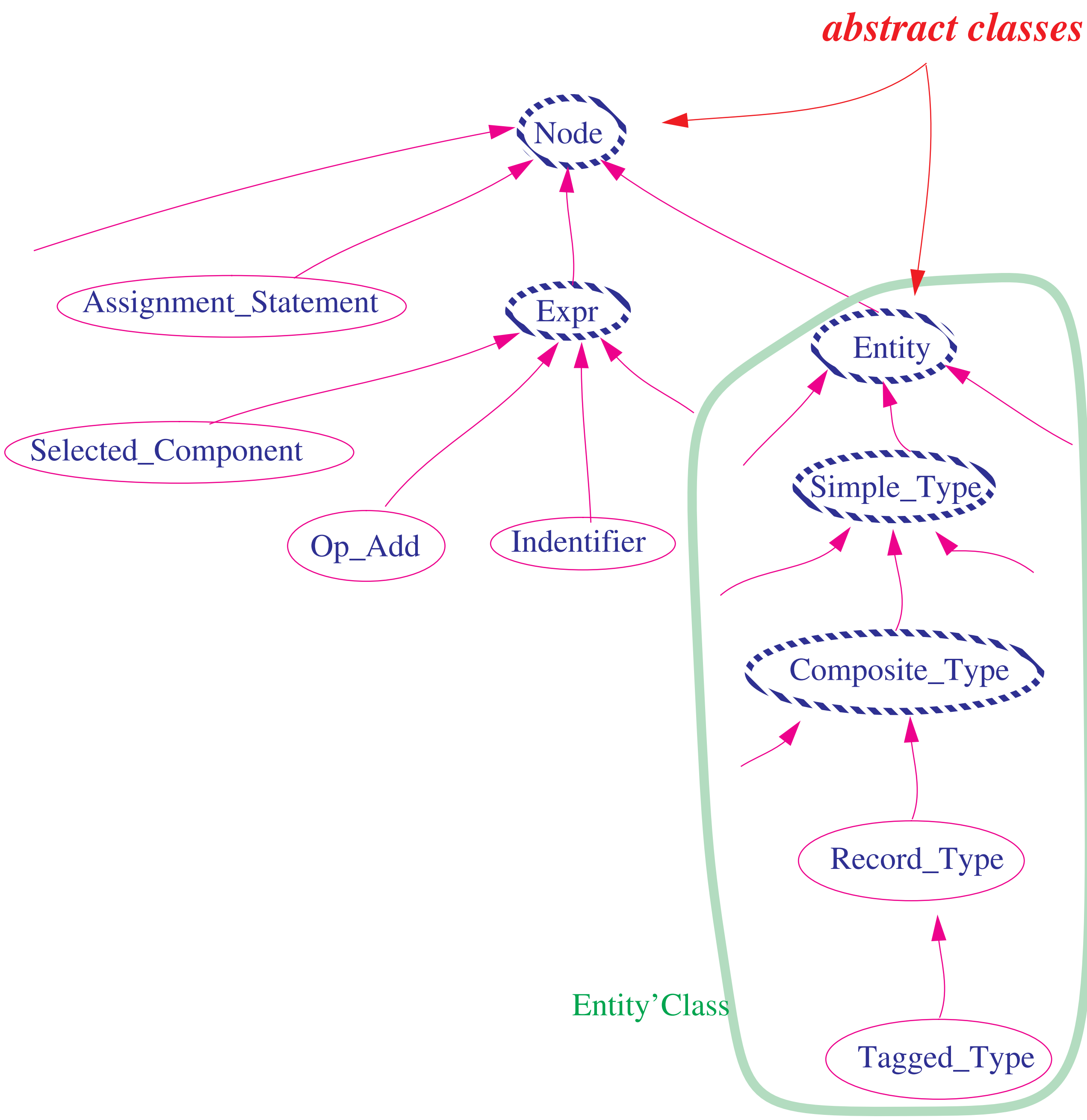
```
procedure Analyze_Node (N : Node_Id) is
begin
   Analyze (N);
   if Code_Generation
    and then N.all not in Expr'Class
   then
      Expand (N);
   end if;
end Analyze_Node;
```

Dispatching Calls:
the formal has a specific tagged type
the actual has a classwide type

```
procedure Analyze (N : access Assignment_Statement) is
   Typ : Type_Id;
begin
   Analyze_Node (N.Name);
   Analyze_Node (N.Expression);
   Typ := Find_Matching_Type (N.Name, N.Expression);
   Resolve_Expr (N.Name, Typ);
   Resolve_Expr (N.Expression, Typ);
end Analyze;
```

This is the specific algorithm for analyzing Assignments

Hierarchy of Tagged Types



Example of recursive definition

```
type Entity;
type Entity_Id is access Entity'Class;


type Entity is abstract new Node with
   record
      Chars : Name_Table_Ptr;
      Etype : Type_Id;
   end record;

procedure Analyze (N : access Entity);
procedure Expand  (N : access Entity);


type Simple_Type is abstract new Entity with
   record
      Init_Proc : Entity_Id;
   end record;
```

Record and Extension Aggregates

```
Rec_Variable.Field := Funct (X) + 1;
```

Record Aggregate

```
Assign_Stmt_Ptr : Node_Id :=

  new Assignement_Statement'(
    Sloc       => Current_Location,
    Name       => new Selected_Component,
    Expression =>

     new Op_Add'(Node with
       Etype      => Standard_Integer,
       Left_Opnd  => new Function_Call,
       Right_Opnd => new Universal_Integer'(Expr with 1)));
```

Extension Aggregates

Syntax

can be of a private type

```
(Ancestor_Expression with [comp_associations])

(Ancestor_Type       with [comp_associations])
```

can be an abstract type

Private Extensions and New Discriminants

new discriminant

```
type Composite_Type (Unconstrained : Boolean) is
   abstract new Simple_Type with null record;
```

discriminant renaming

```
type Record_Type (Discriminated : Boolean) is
   new Composite_Type (Unconstrained => Discriminated)
      with private;
.
.
.
```

private extension

```
type Tagged_Type is new Record_Type with
   record
      Dispatch_Table : Entity_Id;
      Is_Controlled  : Boolean;
   end record;
.
private
```

discriminant inheritance

```
type Record_Type (Discriminated : Boolean) is
   new Composite_Type (Unconstrained => Discriminated)
with
      record
         Components : Entity_List;
         case Discriminated is
            when True  => Discriminants : Entity_List;
            when False => null;
         end case;
      end record;
```

2. Dealing With Common User Mistakes

Find the Mistake ???

case1

```
procedure Error1 is

   type Node is tagged null record;
   type Node_Id is access all Node'Class;
   procedure Analyze (N : access Node) is separate;
   N : Node_Id;

begin
   ...
   Analyze (N);
end Error1;
```

Hints : RM 3.9.2 (9)

> if the expected type of an expression or name is some specific tagged type, then the expression shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation

RM 3.2.3

> The primitive subprograms of a specific type are defined as follows: ... For a specific type declared immediately within a package_Specification, any subprogram that are explicitly declared immediately within the same package-specification and that operate on the type...

What you could have got

```
   Analyze (N);
```

type ‘access to Node’ expected, found Node_Id

```
end Error1;
```

The warning is only issued when an error occurs later

GNAT message

```
procedure Error1 is

   type Node is tagged null record;
   type Node_Id is access all Node'Class;
   procedure Analyze (N : access Node) is separate;
```

warning: not a dispatching operation (must be defined in a package spec)

```
   N : Node_Id;

begin
    Analyze (N);
```

access to class-wide argument not allowed here
“Analyze” is not a primitive operation of “Node”

case2

```
package body Error2 is

   type Ctrl is new Controlled with null record;
   procedure Finalize (Obj : in out Ctrl) is
   begin
       ...
   end Finalize;
end Error2;
```

althrough RM 3.2.3 (7) says :

> The primitive subprograms of a specific type are defined as follows: ... For a specific type declared outside a package_Specification, any subprograms that are explicitly declared immediately within the same declarative region and that override other implicitly declared subprograms of the type

it also says somewhere else (RM 13.14 (15))

> The explicit declaration of a primitive subprogram of a tagged type shall occur before the type is frozen.

and RM 13.14 (4)

> A noninstance body causes freezing of each entity declared before it within the same declarative_part..

What you could have got

```
package body Error2 is

   type Ctrl is new Controlled with null record;
   procedure Finalize (Obj : in out Ctrl) is
```

"Finalization" conflicts with declaration at line 3

GNAT message

```
package body Error2 is

   type Ctrl is new Controlled with null record;
   procedure Finalize (Obj : in out Ctrl) is
```

overriding too late (subprogram spec should appear immediately after type)

```
   begin
      ...
   end Finalize;
end Error2;
```

The problematic notion of Freezing

RM 13.14

...A noninstance body causes freezing ...

...The occurence of an object_declaration... causes freezing...

...The declaration of a record extention causes freezing ...

...At the place where an allocator causes freezing, the designated subtype of its type is frozen

where are the mistakes???

```
package Error3 is

   type Node is tagged null record;
   type Node_Id is access all Node'Class;
   Empty: constant Node_Id := new Node' (null record);
   procedure Analyze (N : access Node);

   type Expr is abstract new Node with null record;
   type Op_Add is new Expr with null record;
   procedure Analyze (N : access Expr) is abstract;
   procedure Analyze (N : access Op_Add);

end Error3;
```

GNAT messages

```
package Error3 is

   type Node is tagged null record;
   type Node_Id is access all Node'Class;
   Empty: constant Node_Id := new Node'(null record);
```

warning: no primitive operations for “Node” after this line

```
   procedure Analyze (N : access Node);
```

this primitive operation is declared too late

```
   type Expr is abstract new Node with null record;
   type Op_Add is new Expr with null record;
```

warning: no primitive operations for “Expr” after this line

```
   procedure Analyze (N : access Expr) is abstract;
```

this primitive operation is declared too late

```
   procedure Analyze (N : access Op_Add);
```

3. Basic Implementation Techniques

Problems that had to be solved:

* Layout of objects

where to put the new discriminants

how to allocate classwide objects

* Structure of the Dispatch Table

The strongly typed approach

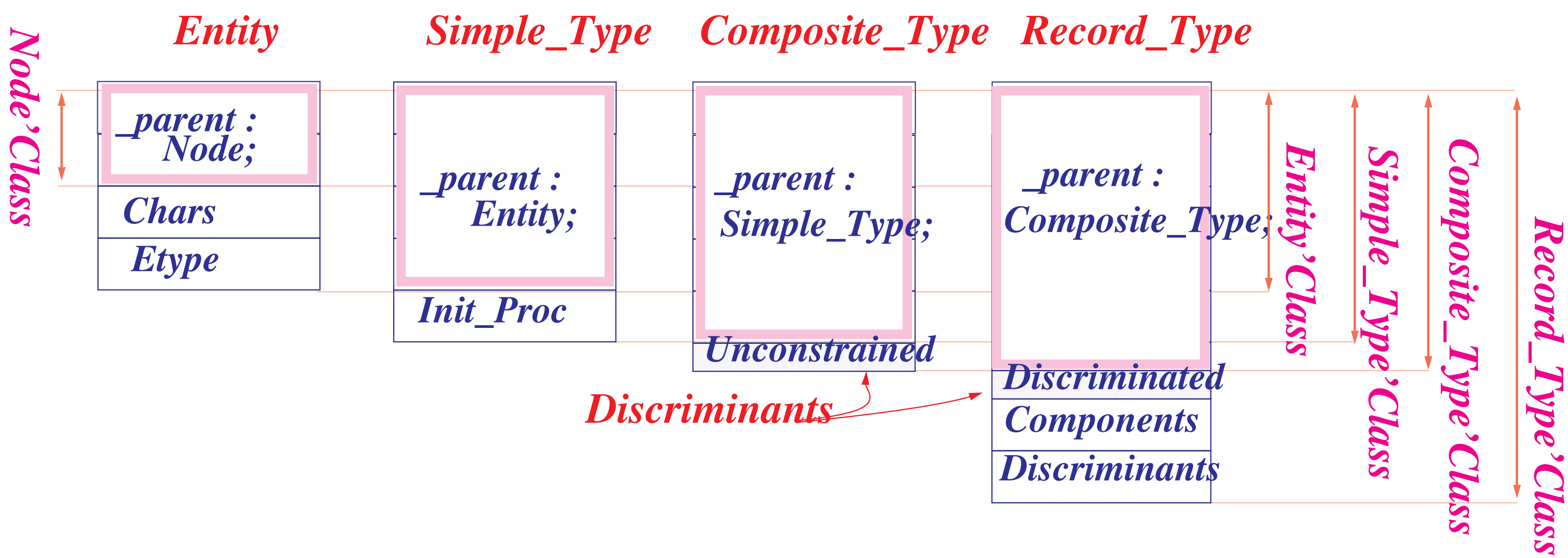
The black box approach

* Membership Test

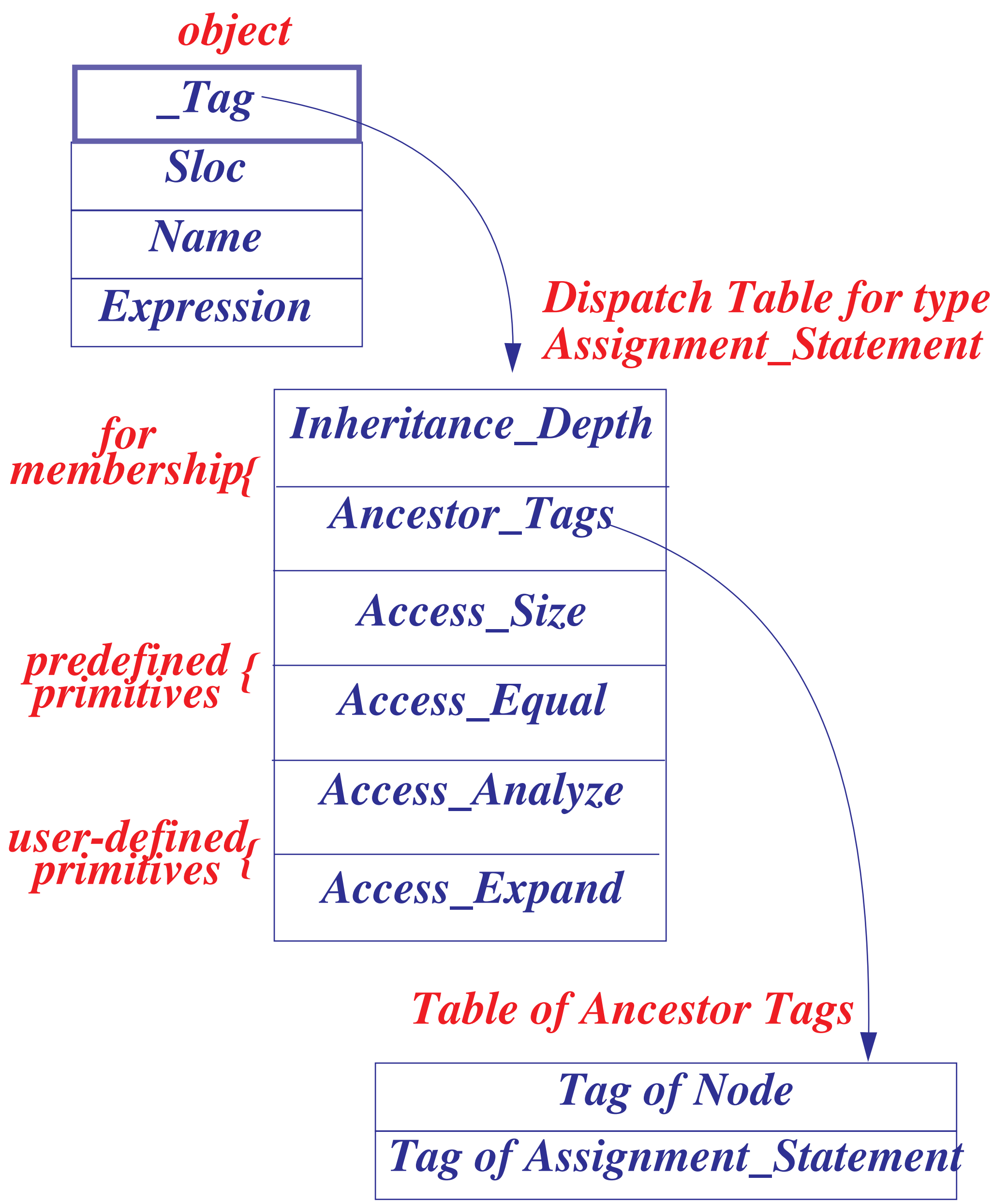
* Visibility issues with private extensions and primitives

Layouts of root tagged types

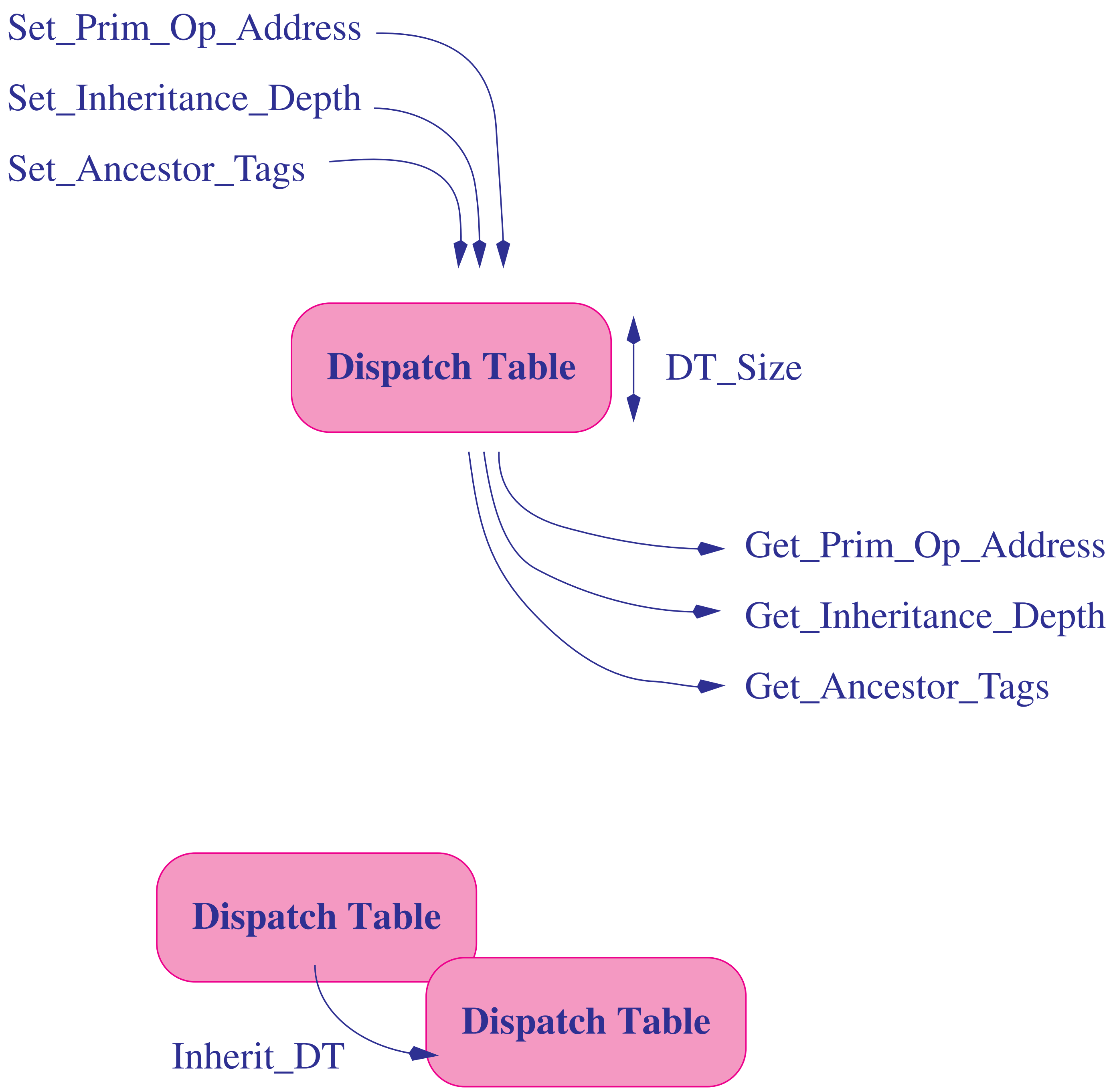
Layouts of extensions



The Strongly Typed View of the Dispatch



The Procedural Interface to the Dispatch



```
package Ada.Tags is

   type Tag is private;

   ...

private

   procedure Set_Prim_Op_Addr (T: Tag; P: Pos; V: Address);
   function  Get_Prim_Op_Addr (T: Tag; P: Pos) return Address;

   procedure Set_Inheritance_Depth (T : Tag; V    : Natural);
   function  Get_Inheritance_Depth (T : Tag) return Natural;

   procedure Set_Ancestor_Tags (T : Tag; V    : Address);
   function  Get_Ancestor_Tags (T : Tag) return Address;

   function DT_Size (C : Natural) return Storage_Count;

   procedure Inherit_DT (OldT, NewT: Tag; C: Natural);

   function CW_Membership (ObjT, TypT : Tag) return Boolean;

   type Dispatch_Table;
   type Tag is access all Dispatch_Table;


   Pragma Inline (Get_Inheritance_Depth);
   Pragma Inline (...);
end Ada.Tags;
```

Taft amendment type

Advantages of Black Box Dispatch Table Approach

The user can modify Ada.Tags without changing the compiler in order to :

- customize the implementation of type **Tag** :

when dispatching is not needed

```
type Tag is null record;
```

```
type Tag is new Array_OF_DT_Index_Type;
```

to trade efficiency for more compact objects

- modify the format of the Dispatch Table

  * to match a **foreign** Dispatch Table **format**

  * to take advantage of a specific architecture

The Low-Level C++ Interface

Goals :

* Import and use C++ Classes as Tagged Types

* Derive from C++ Classes

  dispatching accross languages

  customizing C++ libraries

* Avoid coupling with a specific C++ compiler

Technical problems :

* Semantic differences

* Compatibility of the dispatch tables

* Mangling of names

* Run-time type information

* Layout of objects

Basic Mapping

```
Class C1 {
  public:
    int F (void);
    void P (int v);
    virtual void Disp (void);
    C1 ();

    int v1;
};
```

non-virtual function
=
classwide subprogram

data member
=
record component

constructor
=
initialization function

```
type C1 is tagged record
   V1    : C.Int;
end record;

function   F (This: C1'Class) return C.Int;
procedure P (This: in out C1'Class; V: C.Int);

function   Init return C1'Class;

procedure Disp (This: C1);
```

virtual function
=
primitive operation

To make it work

provide information for dispatching

```
type C1 is tagged record
   V1   : C.Int;
   Vptr : CPP.Vtable_Ptr;
end record;
pragma CPP_Class (C1);
pragma Vtable (C1, Vptr, 1);

function  F (This: C1'Class) return C.Int;
pragma Import (CPP, F, "F", "f_3C1");

procedure P (This: in out C1'Class; V: C.Int);
pragma Import (CPP, P, "P", "p_3C1");

procedure Disp (This: C1);
pragma Import (CPP, Disp, "P", "p_3C1");
pragma CPP_Virtual (Disp, 1);

function  Init return C1'Class;
pragma CPP_Constructor (Init, "C1", "___3C1");
```

mangling is done by hand

Default constructor is automatically applied to CPP objects

The GNAT specific Pragmas

```
pragma CPP_Class (Entity => Typ);
```

Makes the type "limited" disable 'tag, aggregates, membership...

```
pragma CPP_Vtable (
  Entity      => Typ,
  Vtable_Ptr  => Field_Name,
  Entry_Count => Static_Number);
```

tell the compiler where is the Vtable and its size

```
pragma CPP_Virtual (
   Entity     => Subprogram,
   Vtable_Ptr => Field_Name,
   Position   => Static_Number)
```

tell the compiler where is the virtual func.tion in the Vtable

```
pragma CPP_Constructor (
  Entity    => Fname,
  Link_Name => "mangled name");
```

this function can only be used in declaration and allocation

```
pragma CPP_Destructor (
  Entity    => Fname,
  Link_Name => "mangled name");
```

this subprogram will be called on scope exit

```
pragma Import (
  Convention    => CPP,
  Entity        => Entity,
  External_Name => Whatever,
  Link_Name   => "mangled name");
```

no need to provide mangled name for disp. operations

Interfaces.CPP

```
package Interfaces.CPP is

   type Vtable_Ptr is private;

private

   procedure Set_Vfunction_Address (...);
   function  Get_Vfunction_Address (...) return Address;
   procedure Set_Idepth (...);
   function  Get_Idepth (...)              return Natural;
   procedure Set_Ancestor_Vptrs (...);
   function  Get_Ancestor_Vptrs (...)      return Address;
   function  Vtable_Size (...) return Storage_Count;
   procedure Inherit_Vtable (...);
   function  CPP_Membership (...)          return Boolean;

  function Displaced_This (This : Address;
                          Vptr : Vtable_Ptr;
                           Pos  : Positive) return Address;

   type Vtable;
   type Vtable_Ptr is access all Vtable;

   pragma Inline (Get_Vfunction_Address);
   pragma Inline (...);

end Interfaces.CPP;
```

Same structure as Ada.Tags

to accomodate Multiple Inheritance