Part Two

The Core Language

Part One should have given the reader an overall appreciation of the scope and some of the details of Ada 95. The discussion included some rationale for the main features but did not go into all the details. This second part takes the discussion of the Core language a step further. It covers those important features not discussed in Part One and gives more detail of the rationale including alternatives that were considered and rejected. It is assumed that the reader is familiar with the material in Part One which will be referred to from time to time. It is recommended that this part be read in conjunction with the Ada 95 Reference Manual.

1 Introduction

This second part of the rationale is arranged to generally correspond to the sections in the Ada 95 Reference Manual [RM95]. Thus the discussion on exceptions which is covered in section 11 of [RM95] will be found in Chapter 11 of this part. The only exception to this is that the material covered by sections 3 and 4 of [RM95] is subdivided differently. Chapter 3 of this volume covers types and expressions in general whereas Chapter 4 concentrates on the object oriented features such as type extension and class-wide types.

In a similar way the chapters of the third part correspond to the annexes of [RM 95]; thus chapter C discusses the Systems Programming Annex which is annex C of [RM 95].

Each chapter of this second part starts with a brief summary of the changes, there are then a number of sections addressing the various topics, and finally there is a summary of the requirements addressed by the chapter.

This first chapter briefly covers the following general issues

- The description of Ada 95 uses more defined terms and there is less reliance on informal English.
- The syntax is expanded to bring more rules into the syntax and to increase clarity.
- The categorization of errors is revised and includes the introduction of the concept of bounded errors.

However, before getting down to detail it is appropriate to start with a few words about the approach adopted in the development of Ada 95.

1.1 Overall Approach

Ada 95 is based on a building block approach. Rather than providing a number of new language features to directly solve each identified application problem, the extra capability of Ada 95 is provided by a few primitive language building blocks. In combination, these building blocks enable programmers to solve more application problems efficiently and productively.

Thus in the case of the object oriented area, much of the capability is provided by the interweaving of the properties of type extension, the child libraries and generic packages. Great care has been taken to ensure that the additional building blocks work together in a cohesive and helpful manner. Nevertheless implementation and understandability considerations have sometimes caused some restrictions to be imposed. An example is that type extension of a formal generic parameter is not permitted in a generic body (which would either break the contract model or pose an intolerable pervasive implementation burden).

An area of particular difficulty in tasking is the provision of mutual exclusion (which is done in implementation terms by imposing semaphores and locks at appropriate places). Much of the difficulty with Ada 83 tasking lay in the composition of facilities to provide general paradigms and especially the provision of guarded services. Attempts to solve such problems often resulted in race conditions precisely because the facilities did not compose properly. The only solution was paradigm inversion whereby the high level tasking model was used to provide, in essence, some low-level semaphore which could then be used in a medieval fashion. The introduction of protected types, barriers and the requeue statement with two distinct levels of locking is designed to overcome these difficulties. Together, these building blocks may be used to program natural and efficient solutions to problems involving a myriad of real-time paradigms, including asynchronous communication, efficient mutual exclusion, barrier synchronization, counting semaphores, and broadcast of a signal. To have provided separate features to solve each of these problems would have resulted in a baroque language which would have run into difficulties whenever a problem immediately outside the original goals was encountered.

As mentioned in I.3 (in part one), there are four main areas where it was felt that users needed additional functionality: Interfacing to other systems, Programming by Extension (OOP), Programming in the Large (Program Libraries), and Tasking. Broadly speaking these needs are met in Ada 95 by the following main features and are largely discussed in the chapters indicated.

Interfacing: by new forms of access types, pragmas and interface packages (Chapters 3 and B).

Programming by Extension: by type extension and class-wide types (Chapter 4).

Programming in the Large: by child library units (Chapter 10).

Tasking: by protected objects (Chapter 9).

Chapters 3, 4, 9 and 10 constitute the bulk of this part of the rationale mainly because they contain a number of quite long examples. The changes described in the other chapters are more concerned with supporting detail and less pervasive improvements.

1.2 The Reference Manual

The Ada 83 Reference Manual [RM83] is a remarkable document in that it is one of the few definitions of a programming language that is regularly read by normal programmers. It achieves this by using natural English wherever possible.

A corollary of this success is, however, that it has not proved to be quite so precise as desired by compiler writers. Of course, there are many more programmers than compiler writers and so the importance of the programmer should not be underestimated. However, it is vital that the compiler writer be given as precise a description of the language as is reasonably possible. At the end of the day, provided that the compiler is correct, then any misunderstanding of some subtle point on the part of the programmer will generally give rise to an appropriate message from the compiler. Furthermore, textbooks and other material such as this rationale are available to give pedagogic information to the programmer.

The Ada 95 Reference Manual [RM95] thus continues the tradition of readability and accessibility of the Ada 83 document wherever possible but achieves greater precision by the careful introduction of more specific terminology. Different typography is also used to distinguish normal English words from defined terms and syntax thereby increasing clarity but retaining readability.

In addition to the definitive standard, the Annotated Ada Reference Manual [AARM] is an annotated form containing much additional information aimed largely at compiler writers, language lawyers and others with a need for additional detailed information. This contains such matters as advice for implementers, rationale on fine detail, further awkward examples and so on. Both forms of the reference manual as well as this rationale and other material are available in machine readable form on the `sw-eng.falls-church.va.us` host in the `public/ada9x/rm9x` directory.

1.3 Syntax

The syntax is expressed in the same notation as for Ada 83. However, the diligent reader will observe a considerable number of changes to the description of the syntax. Apart from those changes required by the new parts of the language, the changes have been made in order to increase clarity of the exposition.

This increased clarity has been achieved at the cost of introducing rather more new syntax rules than the increased size of the language would suggest. However, the extra clarity achieved brings major benefits not only in understanding the syntax itself but also by some reduction in the need for English text to explain rules which are now expressed by the syntax.

Examples of more notable changes (other than those corresponding to completely new material) are as follows

- The rules for the characters used in the program text have been completely rewritten in a more structured manner. The previous rules were not hierarchical and contained a curious imbalance between upper and lower case letters which is no longer appropriate.

- The category integer is now called numeral. The term integer was somewhat inappropriate for what is simply a syntactic sequence of digits not specifically related to the integer types.

- Reserved words are no longer considered as identifiers. A consequence is that the syntax now explicitly includes those attributes which double as reserved words, namely `Delta`, `Digits`, `Range` and `Access` (the last being a further such attribute in Ada 95).

- Categories such as defining_identifier are introduced for those occurrences of identifiers which define an entity. Usage occurrences use direct_name or selector_name according to the visibility rules. The term simple_name is no longer used. In Ada 83 the term simple_name was used confusingly for just some usage occurrences.

- The category type_declaration now properly includes both task and protected types. Surprisingly, task type declarations were excluded in Ada 83 probably because of a lack of reconsideration of the rules subsequent to the introduction of task types in around 1980.

- The category type_mark is replaced by subtype_mark because all names of types are now considered to actually denote the first named subtype.

- Scalar and composite constraints are now distinguished.

- A real change is that the category name is broadened to include function_call and type_conversion in accordance with changes to the concept of a name. This causes a number of consequential changes to other definitions such as primary and prefix.

- The one previous category aggregate has now been replaced by some nine syntax rules thereby bringing into the syntax the various distinctions between array and record aggregates and their various rules which were previously expressed by English text.

- The new category handled_sequence_of_statements avoids much repetition in a number of other rules, and clarifies the region of text in which a given handler applies.

- The categories body_stub and renaming_declaration are both broken down into named subcategories for ease of exposition.

- The previous category generic_parameter_declaration which confusingly reused other categories from other contexts is now replaced by some twenty individual categories describing the various classes of generic parameters in a hierarchical manner.

The statistically minded might be interested to observe that Ada 83 is described by 180 rules whereas Ada 95 has about 270. However, the rules introduced for clarity account for about 50 of this increase and so in real terms the syntax for Ada 95 is about one seventh bigger than Ada 83. A major part of this increase is simply due to the introduction of protected types.

1.4 Classification of Errors

The classification of errors in Ada 95 is somewhat different to that in Ada 83. The primary reason for the new classification is to be more realistic and practical regarding the possible consequences of undefined behavior. The effect is often to indicate that the range of possible outcomes of a program execution is less than the Ada 83 rules led one to believe (in practice there is little change).

The most significant new classification is the introduction of the category called bounded errors. The general idea is that the behavior is not fully determined but nevertheless falls within well-defined bounds. Many errors which were previously classed as erroneous (which implied completely undefined behavior) are now simply bounded errors. An obvious example is the consequence of evaluating an uninitialized scalar variable; this could result in the raising of `Program_Error` or `Constraint_Error` or the production of a value not in the subtype, see [RM95 4.4, 11.6].

A rather different approach is taken regarding unportable behavior. A program whose behavior depends upon some order of evaluation is no longer classed as incorrect but simply as being not portable. As a consequence the category of error called incorrect order dependences is deleted.

There are also cases where the language has been changed so that a run-time error in Ada 83 is now detected during compilation in Ada 95. Thus static subtype matching is required in some situations as described in 3.9.

The language also allows a compiler to have different modes of operation according to the detection of suspicious situations such as too many warnings. This specifically meets one of the requirements for early detection of errors where possible.

1.5 Requirements Summary

The requirements

R 2.1-A(1) — Incorporate Approved Commentaries

R 2.1-A(2) — Review Other Presentation Suggestions

are both addressed by the extra attention given to terminology and by the incorporation of improved text recommended by the Ada Rapporteur Group.

The requirements

R 2.1-B(1) — Maintain Format of Existing Standard

R 2.1-C(1) — Machine-Readable Version of the Standard

have also been met as explained in 1.2. Furthermore, the requirement

R 2.2-B(1) — Understandability

is also addressed by the greatly improved terminology as well as by the revisions to the syntax described in 1.3. However, it should be noted that, as expressed in [DoD 90], this particular requirement was perhaps slanted more at certain specific language features rather than clarity of description in general.

In the case of error detection the requirement and study topic

R 2.3-A(2) — Limit Consequences of Erroneous Execution

S 2.3-A(1) — Improve Early Detection of Errors

are addressed by the introduction of the concept of bounded errors and more compilation checking such as static subtype matching.

2 Lexical Elements

There are only a small number of changes in this part of the language but they are important. The following are worth noting

- There are six more reserved words.
- The program text character set is extended.
- The rules regarding pragmas are improved.

2.1 Reserved Words and Identifiers

Ada 95 has six more reserved words which are **abstract**, **aliased**, **protected**, **requeue**, **tagged**, and **until**. In addition the word "access" is also used as both an attribute and a reserved word.

The introduction of new reserved words poses a potential incompatibility problem. However, the new words are not likely to have been particularly popular as program identifiers and so little problem should arise.

It was suggested by some reviewers that new reserved words should be avoided by the subtle reuse of existing words in various contexts. This might have led to a bizarre and unnatural language for the sake of avoiding very occasional incompatibility. It would certainly have made the language seem strange and unattractive especially to those not familiar with the background to the development. The smooth integration of the new and important features such as type extension and protected types could not have been achieved without the introduction of additional reserved words.

There are some changes to the terminology in order to clarify the exposition. For example, reserved words are no longer formally classed as identifiers; this has some impact on the syntax as mentioned in 1.3. Also the term numeral is introduced in the discussion of literals.

2.2 Program Text

As part of the original agreement between ISO and ANSI to accept ANSI/MIL-STD-1815A as an international standard, ANSI agreed to provide better support for international character sets in the first revision of Ada.

Therefore, Ada 95 uses an 8-bit character set based on ISO-8859, and a 16-bit character set based on ISO-10646. These extended character sets are usable in character and string literals and also in comments.

The text of a program (outside literals) is typically written in the Latin-1 set, ISO-8859-1 and thereby allows accented characters in identifiers.

Moreover, an implementation is allowed to support other character sets provided that it has a mode in which the "standard" sets are supported. This enables national variations to support sets such as those used in Japan. See [RM95 2.1].

In order to promote portability all implementations are required to support a line length of at least 200 characters.

2.3 Pragmas

In order to improve error detection when dealing with implementation defined pragmas, we require that compilers produce a warning when a pragma is unrecognized, and identify as an error a pragma that is misplaced or malformed. In Ada 83, it was permissible for compilers to ignore such pragmas without a warning, which could lead to unexpected behavior.

We have formalized the definition of configuration pragmas to specify options that affect more than a single compilation unit (often an entire program).

There are a number of additional pragmas in Ada 95 which will be mentioned in the course of the discussion.

2.4 Requirements Summary

The requirement and study topic

R3.1-A(4) — Extended Comment Syntax

S3.1-A(5) — Extended Identifier Syntax

are both addressed and met by the changes mentioned above. Indeed, recently approved Ada Issues permit Ada 83 compilers to meet these requirements on a voluntary (non-binding) basis.

3 Types and Expressions

This chapter covers a number of changes. Some are essentially functional changes in their own right such as the addition of modular types, but many of the changes are more to provide a better framework for the establishment of the object oriented programming facilities which are described in the next chapter. The main changes are

- The foundation concepts and terminology are improved. The idea of a view is formalized. The concepts of a type and class are clarified. The concept of an object is generalized.
- The rules for derived types and their operations are changed to accommodate type extension.
- Character types are changed to conform to the requirements for 8-bit and wider character sets.
- The numeric model is revised to provide a closer mapping to actual machine architectures; the types *root_integer* and *root_real* are added to aid the description.
- Modular (unsigned integer) and decimal fixed point types are added.
- Discriminants are generalized and the concepts of definite and indefinite subtypes introduced. This is particularly relevant to generic parameter matching.
- The rules for implicit array subtype conversion are liberalized to allow sliding in all contexts except qualification (and, of course, membership tests).
- Array aggregates with **`others`** are allowed in more contexts.
- Access types are greatly generalized to provide general access types and access to subprogram types.
- The rules for type conversion are extended to cover the new features in the language.
- The rules for staticness are more liberal thereby allowing more expressions to be considered static.
- There are a number of minor improvements such as the removal of the irritating rule regarding the order of declarations.

It be should be noted that the enhanced forms of access types constitute a major extension to the language in their own right. They provide the more flexible interfacing which was highlighted as one of the four main areas of User Need in I.3.

Type extension, class-wide types and polymorphism are discussed in the next chapter.

3.1 Types, Classes, Objects and Views

The term view is widely used in the description to make it easier to separate properties associated with an entity from properties associated with a particular reference to an entity. For example, a type may have two views, one in places where its full declaration is visible, and one where the type is private. Another example occurs in renaming where two subprogram names may denote the same subprogram, but with different formal parameter names associated with these two different views.

We have generalized the term class to include user-defined classes defined by a type and all its direct and indirect derivatives; we call these derivation classes. The concept of language-defined type classes (such as the discrete class or the real class) allowed the description of Ada 83 to be more economical, and easier to understand. This same economy of definition and understanding is valuable for a user-defined type hierarchy forming a class.

There is a strong distinction between specific types and class-wide types. Specific types are those declared by type declarations, and correspond to Ada 83 types. Each specific tagged type `T` has an associated class-wide type, `T'Class`. Class-wide types enable class-wide (polymorphic) programming, because a subprogram with a formal parameter of a class-wide type like `T'Class` accepts actual parameters for any type covered by the class-wide type (that is, `T` or any of its derivatives). In the implementation of such a subprogram, the operations of the root type (`T` in this case) are available. It is also possible to write dispatching operations, which automatically dispatch to the appropriate implementation based on the type tag of the actual parameter. A class-wide operation of a tagged class-wide type usually calls one or more dispatching operations of the specific type.

The universal types which existed in Ada 83 remain and act much as class-wide types for the numeric classes. However, there are important differences which are discussed in 3.3.

To simplify and unify the description of the Ada 95 type model, we have adopted the terms elementary and composite for describing the two major categories of Ada types. Elementary types have no internal structure, and are used to represent simple values. Composite types are made up of components and other internal state, and are used to represent more complex values and objects. There are a number of existing Ada 83 rules, and new Ada 95 rules, that are made simpler by expressing them only in terms of elementary and composite types, rather than by enumerating more specific type classes.

There was much confusion in Ada 83 regarding the distinction between types and subtypes. In Ada 95, only subtypes have names. A type declaration such as

```
type A is array (Integer range <>) of Float;
```

introduces a first subtype named `A`. The underlying type has no name. In this case the first subtype is unconstrained. (We now say first subtype rather than first named subtype.) On the other hand a declaration such as

```
type B is array (Integer range 1 .. 10) of Float;
```

introduces a first subtype named `B` which is constrained. Another point is that in Ada 83 a type was also a subtype; this is not the case in Ada 95.

This change of nomenclature has no semantic effect; it is designed to simplify later description. In particular, the term type mark is now replaced by subtype mark since it is always the name of a subtype, and one need never say "type or subtype".

The idea of an object is generalized. The result of a function and of evaluating an aggregate are now considered to be (anonymous) constant objects. One consequence of this is that the result of a function call can be renamed; this is particularly useful for limited types, see 7.3. Some things are not objects, for example named numbers are not objects.

3.1.1 Classification of Operations

We have introduced the term primitive operations to encompass that set of operations that are "tightly bound" to a type, being either explicitly or implicitly declared at the point of the type declaration, and inherited by derivatives of the type. These operations are the closed set that effectively define the semantics of the type. The more general term "operation" of a type is no longer formally used.

Ada 83 used "implicit conversion" to explain how integer literals were usable with any integer type, and how real literals were usable with any real type. For Ada 95, we have adopted a similar mechanism as the basis for class-wide programming. However, rather than using the concept of implicit conversion, the static semantic rules are defined in terms of type resolution between actual parameters and formal parameters. (The implicit conversions still happen but are not part of overload resolution.)

As in Ada 83, if the actual parameter and the formal parameter are of the same type, then the actual matches the formal. However, the type resolution rules also allow certain other combinations. In particular, if a formal parameter is of a class-wide type, then the actual parameter may be of any type in the class. This allows the definition of class-wide operations.

A similar approach is taken with universal types. A formal parameter of a universal type is matched by any type of the corresponding numeric class. Thus the `Val` attribute (which accepts an operand of type *universal_integer*) can be matched by any integer type. There is a change to the rules for fixed point multiplication and division which now take *universal_fixed* operands as explained in 3.3.1 and can thus be matched by any fixed point type.

In addition to class-wide matching, the type resolution rules cover the use of access parameters (not to be confused with parameters of an access type, see 3.7.1). When a formal is an access parameter, only the designated type of the actual parameter is considered for matching purposes. The actual matches the formal if their designated types are the same, or, in the case of tagged types, one is `T` while the other is `T'Class`. In addition, for tagged types, changes of representation are not permitted for derived types, so an actual also matches a formal access parameter if the designated type of the actual is covered by the designated type of the formal.

Access parameters allow operations to be defined that take access values rather than designated objects, while still keeping the operation a primitive operation of the designated type. With tagged types, this allows "dispatching on access types" without requiring the access value to be dereferenced first.

Another important change is that the attribute `S'Base` may be used as a subtype mark generally, rather than strictly as a prefix for other attributes. `S'Base` denotes an unconstrained subtype of the type of `S` and is only allowed for elementary types. It is particularly useful within a generic package that might be instantiated with a constrained numeric subtype, since the temporary variables used to perform a calculation might need to be unconstrained, even if the parameters and final result of an operation must satisfy the constraints of the actual constrained subtype.

For example consider the implementation of `Generic_Elementary_Functions`. We need to allow the user to instantiate the package with a constrained subtype corresponding to `Float_Type`, but do not wish the calculations to be constrained. Accordingly the parameters and results of the various functions are of the subtype `Float_Type'Base`.

One potential problem with allowing the declaration of objects of subtype `S'Base` is that the first subtype (for example `S`) may have a size clause that takes advantage of the constraints on `S`. Objects of subtype `S'Base` cannot generally be limited by the size specified for `S`. There are several reasons why this problem is not serious in practice:

- Many compilers already use different sizes for different subtypes of the same type;

- The construct **`for`** `B` **`in`** `S'Base'First .. S'Base'Last` **`loop ...`** is already legal in Ada-83 (presuming `S` is discrete), and is an existing way to effectively create an object (`B`) of subtype `S'Base`;

3.1.2 Derived Types

For Ada 95, we have chosen to build upon the Ada 83 derived type mechanism to provide for type extension (single inheritance) and run-time polymorphism, two fundamental features of object-oriented programming. (Derived types were the existing type inheritance mechanism in Ada 83.) If a new inheritance mechanism had been introduced, perhaps based on "package types" or an explicit "class" construct, inheritance based on derived types would still remain as an almost redundant and complicating alternative inheritance mechanism. Choosing to enhance the basic derived type mechanism provides a single robust inheritance mechanism rather than two potentially conflicting and weaker ones.

Rather than introducing an explicit class construct, we have instead chosen to support user-defined classes via a hierarchy of derived types. The (derivation) class rooted at a type `T` consists of `T` and all of its direct and indirect derivatives. The existing Ada 83 rules for derived types ensure that all of the types in the class rooted at `T` have at least the same set of primitive operations as `T`, because a derivative may override and add operations, but it cannot eliminate an operation inherited from the parent type.

Having a set of operations that are well defined for all types in a class rooted at some type `T` makes it meaningful to construct class-wide operations that take advantage of this commonality. Much of the power and economy of object-oriented programming comes from the ability to write such class-wide operations easily.

If an operation is explicitly defined on a class-wide type, then it is a class-wide operation via the type resolution rules.

The existing universal types behave very much as class-wide numeric types. In fact we introduce types *root_integer* and *root_real* as the numeric types from which all other numeric types are descended and then the universal types can be considered to be the class-wide types corresponding to these root types.

Ada 83 already had existing operations such as the `Val` attribute that took an operand of any integer type; in Ada 95 this is described by saying that `Val` takes an operand of the *universal_integer* type. These are therefore like class-wide operations.

3.2 Character Types

We mentioned in Chapter 2 that the text of an Ada 95 program can be written using more liberal character sets. In this section we consider the support for character types in the executing program.

As part of providing better support for international character sets, the fundamental character set of Ada 95 is changed from the seven-bit ISO 646 standard, to the eight-bit ISO 8859 standard (which includes Latin-1). This means that the type `Character` in package `Standard` is now an enumeration type with 256 positions, rather than just 128.

This change is not upward compatible for programs that have arrays indexed by `Character`, or case statements over `Character`. However, the benefits of accommodating international character sets were felt to outweigh the costs of this upward incompatibility. See X.2.

To facilitate direct use of character literals and string literals from all languages in the international community, a type `Wide_Character` is declared in package `Standard`. The type `Wide_Character` has `2**16` positions, and starts with the 256 enumeration literals of the type `Character`.

The predefined library package `Ada.Characters` has a child package `Characters.Handling` containing useful classification and conversion functions (such as `Is_Letter` and `To_Lower`) and a child package `Characters.Latin_1` containing constants for the Latin-1 symbol set.

There is also a string type `Wide_String` indexed by subtype `Positive`, with component subtype `Wide_Character`.

3.3 Numeric Types

The model of numeric types is somewhat different in Ada 95. The overall goal of the change is to give the implementation more freedom for optimizations such as keeping intermediate results and local variables in registers. Most of the change is fine detail that need not concern the normal user and is addressed in the Numerics annex. However, one area that is important in the core language is the somewhat different treatment of universal types and the introduction of the anonymous types *root_integer* and *root_real*.

The essence of the root types is that they can be considered as the types from which all other integer and real types are derived. The base range of *root_integer* is thus `System.Min_Int .. System.Max_Int`. We will first discuss the integer types and then indicate where the floating types differ.

We have introduced the term base range for the implemented range of a type whereas range refers to the requested range of a particular subtype. Range checks *only* apply to constrained subtypes; overflow checks *always* apply. An important consequence is that we either get the mathematically correct answer or `Constraint_Error` is raised.

Thus if we write

```
type My_Integer is range -20_000 .. +20_000;
MI: My_Integer;
MIB: My_Integer'Base;
```

then `My_Integer'Range` will be `-20_000 .. +20_000` and all assignments to variables of the subtype `My_Integer` such as `MI` will be checked to ensure that the range is not violated; `Constraint_Error` is raised if the check fails.

On the other hand, the base range of `My_Integer` is the range of `My_Integer'Base` and this will be that of the implemented range which might reflect that of a 16-bit word and thus be `-2**15 .. +2**15-1`. No range checks apply to assignments to the variable `MIB`. However, as an optimization, it might be the case that a particular variable of the subtype `My_Integer'Base` is held in a register and this could have a wider range than the base range of the subtype. The base range is thus the guaranteed minimum implemented range. Nevertheless overflow checks will always apply and `MIB` will never have a mathematically incorrect value although the value could be outside the base range. For example, consider

```
X: My_Integer := 18_000;
Y: My_Integer := 15_000;
...
MIB := X + Y;
```

where we will assume that the computation is not all optimized away by a smart compiler!

(Note that no explicit conversion is needed because `My_Integer` and `My_Integer'Base` are both subtypes of the same (unnamed) type. Remember that all types are unnamed.)

If `MIB` is implemented with its base range then an overflow will occur and result in `Constraint_Error` because the result is outside the base range. If, however, `MIB` is held in a 32-bit register, then no overflow will occur and `MIB` will have the mathematically correct result. On the other hand

```
MI := X + Y;
```

will always result in `Constraint_Error` being raised because of the range check upon the assignment.

In the case of the predefined types such as `Integer` the same rules apply; the subtype `Integer` is constrained whereas `Integer'Base` is not. The base range and range happen to be the same. So the declarations

```
I: Integer;
IB: Integer'Base;
```

have a different effect. Checks will apply to assignments to `I` but not to assignments to `IB` (but remember that an implementation is always free to add checks if convenient; they may be automatic).

Another possibility for optimization is that an intermediate expression might be computed with a larger range. This is why the predefined operators such as `"+"` on the predefined types such as `Integer` have parameters and result of `Integer'Base` rather than `Integer`. There are no range checks on these operations (just overflow checks). Now consider

```
MI := X * Y / 30_000;
```

in which we will assume that the computation is done with the operations of type `Integer` which has a 16-bit base range on this implementation. If the operations are done from left to right and the operations are performed in 16-bit registers then overflow will occur and `Constraint_Error` will be raised. On the other hand, the operations might be performed in 32-bit registers in which case overflow will not occur and the correct result will be assigned to `MI` after successfully performing a range check on the result.

The universal types are types which can be matched by any specific numeric type of their class. We see therefore that the universal types are rather like class-wide types of the respective classes. So *universal_integer* is thus effectively *root_integer*`'Class`.

The integer literals are, of course, of the type *universal_integer* and so, as in Ada 83, can be implicitly converted to any integer type including the anonymous *root_integer*. An important distinction between universal and tagged class-wide types is that the latter carry a tag and explicit conversion to a specific type is required which is checked at runtime to ensure that the tag is appropriate, see 3.8.

One consequence of treating *universal_integer* as matching any integer type is that the rules for the initial expression in a number declaration are more liberal than they were in Ada 83. The initial expression can be of any integer type whereas in Ada 83 it had to be universal; it still of course has to be static.

Similar remarks apply to real types. In the case of floating point types a range check is only applied if the definition contains a range (this is the same rule as for integer type definitions but they always have **range** anyway). So given

```
type My_Float is digits 7;
type Your_Float is digits 7 range -1.0E-20 .. +1.0E+20;
```

then `My_Float` is an unconstrained subtype whereas `Your_Float` is constrained. Range checks will apply on assignments to `Your_Float` but not to `My_Float`. The predefined types such as `Float` are unconstrained; it is considered that their notional definition does not include a range.

Overflow checks apply to floating point computations only if the attribute `Machine_Overflows` is true as in Ada 83.

By introducing root numeric types, the special Ada 83 rules regarding convertible universal operands are eliminated (only certain simple expressions could be automatically converted in Ada 83). Instead, the distinction between convertible and non-convertible universal operands corresponds directly to the distinction between the universal and specific root numeric types. The operators of the root numeric types return specific root numeric types, and hence their result is not universal (not "implicitly convertible" using Ada 83 terminology). The type resolution rules ensure that these operators accept operands of the universal types, so they may be used on literals and named numbers.

There is an important change to the visibility rules concerning a preference for the root types in the case of an ambiguity. This is discussed in 8.4.

In order to promote precise use of specific hardware the library package `Interfaces` defines signed integer types corresponding to the hardware supported types with names such as

`Integer_32` and `Integer_16` plus corresponding modular types (see 3.3.2). This package also predefines similar floating types corresponding to the hardware although no names are prescribed.

The description of the real numbers is greatly simplified. The model and safe numbers of Ada 83 have been abandoned because they were not well understood, did not truly provide the portability they sought and obscured the real machine from the specialist. Accordingly the definition of floating point is now in terms of model numbers which roughly correspond to the old safe numbers and are close to the represented numbers. In the case of fixed point the definition is entirely in terms of *small* and the notion of model numbers no longer applies.

To avoid confusion and to improve the correspondence between the real type attributes and the machine attributes, the attributes are completely redefined so that they more closely correspond to the capabilities of the machine.

The description of model numbers is moved to the Numerics annex because of its specialist nature. For more details consult Part Three of this rationale.

We considered removing floating point and fixed point accuracy constraints from the syntax so that delta and digits would only be specified as part of a real type definition. Indeed, AI-571 concluded that reduced accuracy real subtypes should not be represented with reduced accuracy, making their usefulness in the language questionable. However, they are retained (although considered obsolete) for compatibility because of the different format obtained with `Text_IO`.

3.3.1 Operations

The mixed multiplying operators of the Ada 83 universal numeric types are redefined in Ada 95 in terms of the root numeric types.

There were some essentially unnecessary restrictions on the use of literals in fixed point multiplication and division in Ada 83. These operations now take *universal_fixed* as their operands and return *universal_fixed* as the result. However the result must be in a context which provides a specific expected type. As a consequence literals may now be used more freely in fixed point operations and a multiplication or division need not be followed by conversion to a specific type if the context supplies such a type.

So given two fixed point types `Fixed1` and `Fixed2`, we can now write sequences such as

```
X, Y: Fixed1;
Z: Fixed2;
...
X := 2.0 * X;
X := Y * Z;
```

which were forbidden in Ada 83. Note that multiple operations as in

```
X := X * Y / Z;
```

remain forbidden since the context does not provide a type (and therefore an accuracy and range) for the intermediate result.

3.3.2 Modular Types

In Ada 95 the integer types are subdivided into signed integer types and modular types. The signed integer types are those with which we are already familiar from Ada 83 such as `Integer` and so on. The modular types are new to Ada 95.

The modular types are unsigned integer types which exhibit cyclic arithmetic. (They thus correspond to the unsigned types of some other languages such as C.) A strong need has been felt for some form of unsigned integer types in Ada and most compiler vendors have provided their own distinct implementations. This of course has caused an unnecessary lack of portability which the introduction of modular types in Ada 95 will overcome.

As an example consider unsigned 8-bit arithmetic (that is byte arithmetic). We can declare

```
type Unsigned_Byte is mod 256;  -- or mod 2**8;
```

and then the range of values supported by `Unsigned_Byte` is `0 .. 255`. The normal arithmetic operations apply but all arithmetic is performed modulo 256 and overflow cannot occur.

The modulus of a modular type need not be a power of two although it often will be. It might, however, be convenient to use some obscure prime number as the modulus perhaps in the implementation of hash tables.

The logical operations **and**, **or**, **xor** and **not** are also available on modular types; the binary operations naturally treat the values as bit patterns; the **not** operation subtracts the value from its maximum. No problems arise with mixing these logical operations with arithmetic operations because negative values are not involved.

The logical operations will be most useful if the modulus is a power of two; they are well defined for other moduli but there are some surprising effects. For example DeMorgan's theorem that

```
not(A and B) = not A or not B
```

does not hold if the modulus is not a power of two.

The package `Interfaces` defines modular types corresponding to each predefined signed integer type with names such as `Unsigned_16` and `Unsigned_32`. For these modular types (which inevitably have a modulus which is a power of two) a number of shift and rotate operations are also provided.

It is an important principle that conversion between numeric types should not change the value (other than rounding). Conversion from modular to signed integer types and vice versa is thus allowed provided the value is in the range of the destination; if it is not then `Constraint_Error` is raised.

Thus suppose we had

```
type Signed_Byte is range -128 .. +127;
U: Unsigned_Byte := 150;
S: Signed_Byte := Signed_Byte(U);
```

then `Constraint_Error` will be raised.

Unchecked conversion can be used to convert patterns out of range. We could neatly write

```
function Convert_Byte is
   new Unchecked_Conversion(Signed_Byte, Unsigned_Byte);
function Convert_Byte is
   new Unchecked_Conversion(Unsigned_Byte, Signed_Byte);
```

providing conversions in both directions and then

```
S := Convert_Byte(U);
```

would result in `S` having the value `-106`.

The modular types form a distinct class of types to the signed integer types. There is thus a distinct form for a generic formal parameter of a modular type namely

```
type T is mod <>;
```

and this cannot be matched by a type such as `Integer`. Nor indeed can the signed integer form with **range** <> be matched by a modular type such as `Unsigned_32`.

The new attribute `Modulus` applies to a modular type and returns its modulus. This is of particular value with generic parameters.

3.3.3 Decimal Types

Decimal types are used in specialized commercial applications and are dealt with in depth in the Information Systems annex. However, the basic syntax of decimal types is in the core language.

A decimal type is a form of fixed point type. The declaration provides a value of delta as for an ordinary fixed point type (except that in this case it must be a power of 10) and also prescribes the number of significant decimal digits. So we can write

```
type Money is delta 0.01 digits 18;
```

which will cope with values of a typical decimal currency. This allows 2 digits for the cents and 16 for the dollars so that the maximum allowed value is

9,999,999,999,999,999.99

The usual operations apply to decimal types as to other fixed point types. Furthermore the Information Systems annex describes a number of special packages for decimal types including conversion to human readable output using picture strings.

Much as with modular types there is also a special form for a generic parameter of a decimal type which is

```
type T is delta <> digits <>;
```

This cannot be matched by an ordinary fixed point type and similarly the form with just **delta** <> cannot be matched by a decimal type such as `Money`.

3.4 Composite Types

In Ada 95, the concept of composite types is broadened to include task and protected types. This is partly a presentation issue and partly reflects the generalization of the semantics to allow discriminants on task and protected types as well as on records.

The terms definite and indefinite subtypes are introduced as explained in II.11 when we discussed the generic parameter mechanism. Recall that a definite subtype is one for which an uninitialized object can be declared such as `Integer` or a constrained array subtype or a record subtype with discriminants with defaults. An indefinite subtype is an unconstrained array subtype or an unconstrained record, protected or task subtype which does not have defaults for the discriminants, or a class-wide subtype or a subtype with unknown discriminants.

As a simple generalization to Ada 83, we have allowed both variables and constants of an indefinite subtype to be declared, so long as an initial value is specified; the object then takes its bounds or discriminants from the initial value. In Ada 83, only initialized constants of such a subtype could be declared. However, the implementation considerations are essentially identical for constants and variables, so eliminating the restriction against variables imposes no extra implementation burden, and simplifies the model.

Here is an example of use

```
if Answer /= Correct_Answer then
   declare
      Image: String := Answer_Enum'Image(Correct_Answer);
   begin
      Set_To_Lower_Case(Image);
```

```
        Put_Line("The correct answer is " & Image & '.');
    end;
end if;
```

Allowing composite variables without a specified constraint to be declared, if initialized, is particularly important for class-wide types and (formal) private types with discriminant part `(<>)` since such types have an unknown set of discriminants and thus cannot be constrained. For example, in the case of class-wide types, it would otherwise be hard if not impossible to write the procedure `Convert` in 4.4.3 since we would not be able to declare the temporary variable `Temp`.

3.4.1 Discriminants

A private type can now be marked as having an unknown number of discriminants thus

```
type T(<>) is private;
```

The main impact of this is that the partial view does not allow uninitialized objects to be declared. If the partial view is also limited then objects cannot be declared at all (since they cannot be initialized). The gives the writer of an abstraction rather more control over the use of the abstraction.

As we have already noted, discriminants are also allowed on task and protected types in Ada 95. (An early draft of Ada 9X also permitted discriminants on arrays and discriminants to be of any nonlimited type. This was, however, felt to be too much of a burden for existing implementations.)

Discriminants are the primary means of parameterizing a type in Ada, and therefore we have tried to make them as general as possible within transition constraints. Task and protected types in particular benefit from discriminants acting as more general type parameters.

In Ada 83, an instance of a task type had to go through an initial rendezvous to get parameters to control its execution. In Ada 95, the parameters may be supplied as discriminant values at the task object declaration, eliminating the need for the extra rendezvous. Variables introduced in the declarative part of the task body can also depend on the task type discriminants, as can the expression defining the initial priority of the task via a `Priority` pragma. See 9.6 for some detailed examples.

In addition to allowing discrete types as discriminants as in Ada 83, we now also permit discriminants to be of an access type. There are two quite distinct situations. A discriminant can be of a named access type or it can be an access discriminant in which case the type is anonymous. Thus we can declare

```
type R1(D: access T) is ...
type AT is access T;
type R2(D: AT) is ...
```

and then the discriminant of `R1` is an access discriminant whereas the discriminant of `R2` is a discriminant of a named access type. A similar nomenclature applies to subprogram parameters which can be access parameters (without a type name) or simply parameters of a named access type (which was allowed in Ada 83).

Access discriminants provide several important capabilities. Because they impose minimal accessibility checking restrictions, an access discriminant may be initialized to refer to an enclosing object, or to refer to another object of at least the same lifetime as the object containing the discriminant. Access discriminants can only be applied to limited types. Note also that a task and a protected object can have access discriminants.

When an object might be on multiple linked lists, it is typical that one link points to the next link. However, it is also essential to be able to gain access to the object enclosing the link as well.

With access discriminants, this reference from a component that is a link on a chain, to the enclosing object, can be initialized as part of the default initialization of the link component. This is discussed further in 4.6.3. For a fuller discussion on how access discriminants avoid accessibility problems see 3.7.1. Further examples of the use of access discriminants will be found in 7.4 and 9.6.

Finally, a derived type may specify a new set of discriminants. For untagged types, these new discriminants are not considered an extension of the original type, but rather act as renamings or constraints on the original discriminants. As such, these discriminants must be used to specify the value of one of the original discriminants of the parent type. The new discriminant is tightly linked to the parent's discriminant it specifies, since on conversion from the parent type, the new discriminant takes its value from that discriminant (presuming `Constraint_Error` is not raised). The implementation model is that the new discriminants occupy the space of the old. The new type could actually have less discriminants than the old. The following are possible

```
type S1(I: Integer) is ...;
type S2(I: Integer; J: Integer) is ...;

type T1(N: Integer) is new S1(N);
type T2(N: Integer) is new S2(N, 37);
type T3(N: Integer) is new S2(N, N);
```

The last case is interesting because the new discriminant is mapped onto both the old ones. A conversion from type `S2` to `T3` checks that both discriminants of the `S2` value are the same. A practical use of new discriminants for non-tagged types is so that we can make use of an existing type for the full type corresponding to a private type with discriminants.

```
   type T(D: DT) is private;
private
   type T(D: DT) is new S(D);
```

In the case of a tagged type, we can either inherit all the discriminants or provide a completely new set. In the latter case the parent must be constrained and the new discriminants can (but need not) be used to supply the constraints.

Thus a type extension can have more discriminants than its parent, which is not true in the untagged case.

3.5 Array Types

A very minor change is that an index specification of an anonymous array type in an initialized declaration can also take the unconstrained form

```
V: array (Integer range <>) of Float :=
                                        (3 .. 5 => 1.0, 6 | 7 => 2.0);
```

in which case the bounds are deduced from the initial expression.

3.5.1 Array Aggregates

Ada 83 had a rule that determined where a named array aggregate with an **others** choice was permitted; see [RM83 4.3.2(6)]. There were a related set of rules that governed where implicit array subtype conversion ("sliding") was permitted for an array value; see [RM83 3.2.1(16) and 5.2.1(1)]. These rules were constructed to ensure that named array aggregates with others and array sliding were not both permitted in the same context. However, the lack of array sliding in certain contexts could result in the unanticipated raising of `Constraint_Error` because the bounds did not match the applicable constraint.

For Ada 95, we have relaxed the restrictions on both array sliding and named array aggregates with others, so that both are permitted in all contexts where an array aggregate with just an others choice was legal in Ada 83. This corresponds to all situations where an expression of the array type was permitted, and there was an applicable index constraint; see [RM83 4.3.2(4-8)]. This ensures that sliding takes place as necessary to avoid `Constraint_Error`, and simplifies the rules on array aggregates with an others choice.

The original Ada 83 restrictions were related to the possible ambiguity between determining the bounds of an aggregate and sliding. In Ada 95, this ambiguity is resolved by stipulating that sliding never takes place on an array aggregate with an others choice. The applicable index constraint determines the bounds of the aggregate.

As an example consider

```
type Vector is array (Integer range <>) of Float;
V: Vector(1 .. 5) := (3 .. 5 => 1.0, 6 | 7 => 2.0);
```

which shows a named aggregate being assigned to `V`. The bounds of the named aggregate are `3` and `7` and the assignment causes the aggregate to slide with the net result that the components `V(1) .. V(3)` have the value `1.0` and `V(4)` and `V(5)` have the value `2.0`.

On the other hand writing

```
V := (3 .. 5 => 1.0, others => 2.0);
```

has a rather different effect. It was not allowed in Ada 83 but in Ada 95 has the effect of setting `V(3) .. V(5)` to `1.0` and `V(1)` and `V(2)` to `2.0`. The point is that the bounds of the aggregate are taken from the context and there is no sliding. Aggregates with **`others`** never slide.

Similarly no sliding occurs in

```
V := (1.0, 1.0, 1.0, others => 2.0);
```

and this results in setting `V(1) .. V(3)` to `1.0` and `V(4)` and `V(5)` to `2.0`.

3.5.2 Concatenation

The rules for concatenate (we now use this more familiar term rather than catenate) are changed so that it works usefully in the case of arrays with a constrained first subtype.

In Ada 83 the following raised `Constraint_Error`, while in Ada 95 it produces the desired result

```
   X: array (1..10) of Integer;
begin
   X := X(6..10) & X(1..5);
```

In Ada 83, the bounds of the result of the concatenate were `6 .. 15`, which caused `Constraint_Error` to be raised since 15 is greater than the upper bound of the index subtype. In Ada 95, the lower bound of the result (in this constrained case) is the lower bound of the index subtype so the bounds of the result are `1 .. 10`, as required.

3.6 Record Types

A record type may be specified as tagged or limited (or both) in its definition. This makes record types consistent with private types, and allows a tagged record type to be declared limited even if none of its components are limited. This is important because only limited types can be extended with components that are limited.

A derived type is a record extension if it includes a record extension part, which has the same syntax as a normal record type definition preceded by the reserved word **with**. For example

```
type Labelled_Window(Length : Natural) is new Window with
   record
     Label: String(1..Length);
   end record;
```

Record extension is the fundamental type extension (type inheritance) mechanism in Ada 95. A private extension must be defined in terms of a record extension. The new discriminants in a discriminant extension are normally used to control the new components defined in the record extension part (as illustrated in the above example).

Record extension is a natural evolution of the Ada 83 concept of derived types. From an implementation perspective, it is relatively straightforward, since the new components may all be simply added at the end of the record, after the components inherited from the parent type.

We considered having other kinds of type extension, including enumeration type extension, task type extension, and protected type extension. However, none of these seemed clearly as useful as record extension, and all introduced additional implementation complexities. In any case, the automatic assignment of tags to type extensions lessens the need for enumeration types, and the added flexibility associated with access to subprogram and dispatching operations makes it less critical to allow task types to be extended.

Type extension of protected objects was another interesting possibility. However, certain implementation approaches do not easily support extension of the set of protected operations, or the changing of the barrier expressions. With some regret therefore it was decided that the benefit of extending protected types was not worth the considerable implementation burden. This is an obvious topic for review at the next revision of Ada.

Type extension is only permitted if the parent type is tagged. Originally we considered allowing any record or private type to be extended, but this introduced additional complexity, particularly inside generics. Furthermore, extending an untagged type breaks the general model that a class-wide type can faithfully represent any value in the class. An object of an untagged class-wide type would not have any provision for holding a value of a type extension, since it would lack a run-time type tag to describe the value.

3.6.1 Record Aggregates

Record aggregates are only permitted for a type extension if both the extension part and the parent part are fully visible. This corresponds to the principle that if part of a type is private, then it must be assumed to have an unknown set of components in that part. In other words we can only use an aggregate where we can view all the components.

However, extension aggregates can be used provided only that the components in the extension part are visible; we do not need a full view of the type of the ancestor expression. Typically we can provide an expression for the ancestor part. Thus suppose we have

```
type T is tagged private;
...
T_Obj: T := ...;
type NT is new T with
   record
      I, J: Integer;
   end record;
```

then we can write an extension aggregate such as

```
(T_Obj with I => 10, J => 20)
```

A variation is that we can also simply give the subtype name as the ancestor part thus

```
(T with I => 10, J => 20)
```

which is essentially equivalent to declaring a temporary default initialized object of the type and then using it as the ancestor expression (this includes calling `Initialize` in the case of a controlled type, see 7.4). This is allowed even if the ancestor type is abstract and thereby permits the creation of aggregates for types derived from abstract types.

3.6.2 Abstract Types and Subprograms

As we have already discussed in II.3, a tagged type may be declared as abstract by the appearance of **abstract** in its declaration. A subprogram which is a primitive operation of an abstract tagged type may be specified as abstract. An abstract subprogram has no body, and cannot be called directly or indirectly. A dispatching call will always call some subprogram body that overrides the abstract one because it is not possible to create an object of an abstract type.

If a type is derived from an abstract type and not declared as abstract then any inherited abstract subprograms must be overridden with proper subprograms. Note, of course, that an abstract type need not have any abstract subprograms.

The interaction between abstract types and private types is interesting. It will usually be the case that both views are abstract or not abstract. However, it is possible for a partial view to be abstract and the full view not to be abstract thus

```
package P is
   type T is abstract tagged private;
private
   type T is tagged ...;
end P;
```

In this case, objects of the type can only be declared for the full view and abstract primitive operations cannot be declared at all. This is because an abstract operation in the visible part would still apply in the private part and would thus be abstract for the nonabstract view.

It is of course not possible for the full view to be abstract and the partial view not to be abstract. This is quite similar to the rules for limitedness. A partial view can be limited and a full view not limited but not vice versa; the key point is that the partial view cannot promise more properties than the full view (the truth) actually has.

Of more interest is private extension where again the partial view could be declared abstract and the full view not abstract. An inherited abstract subprogram would need to be overridden. If this were done in the private part then the partial view of the subprogram would still be abstract although the full view would be of the overriding subprogram and thus not abstract. Thus

```
package P is
   type T is abstract tagged null record;
   procedure Op(X: T) is abstract;
end P;

with P;
package NP is
   type NT is abstract new P.T with private;
private
   type NT is new T with ...;
   procedure Op(X: T);    -- overrides
end NP;
```

The overriding is essential since otherwise we might dispatch to an abstract operation.

Another point is that an abstract type is not allowed to have an invisible abstract operation since otherwise it could not be overridden. The following difficulty is thus avoided.

```
package P is
   type T is abstract ...;
   procedure Nasty(X: T'Class);
private
   procedure Op(X: T) is abstract;  -- illegal
end P;

package body P is
   procedure Nasty(X: T'Class) is
   begin
      Op(X);
   end Nasty;
end P;

with P;
package Q is
   type NT is new P.T with ...;   -- not abstract
   -- cannot see Op in order to override it
end Q;
```

The problem is that we must override `Op` since by declaring an object of type `NT` we can then dispatch to `Op` by calling the procedure `Nasty`.

The overall motivation for the rules is to ensure that it is never possible to dispatch to a non-existent subprogram body.

A rather different problem arises when we extend a type which is not abstract but which has a function with a controlling result. The old function cannot be used as the inherited version because it cannot provide values for the type extension when returning the result (parameters are not a problem because they only involve conversion towards the root). As a consequence the type must be declared as abstract unless we provide a new function.

A related restriction is that a function with a controlling result cannot be declared as a private operation since otherwise a similar difficulty to that discussed above would arise on type extension. If extension were performed using the partial view then the function would become abstract for the extended type and yet, being private, could not be overridden.

Observe also that since we do not require every abstract subprogram to be explicit, it is possible for a generic package specification to define an abstract record extension of a formal tagged type without knowing exactly which functions with controlling results exist for the actual type.

Finally, note that it is possible to have an abstract operation of a nontagged type. This is fairly useless since dispatching is not possible and static calls are illegal. However, it would be harder to formulate the rules to avoid this largely because an operation can be primitive of both a tagged and nontagged type (although not of two tagged types, see 4.5).

3.7 Access Types

As we have already seen in II.5 and II.6, access types in Ada 95 have been generalized so that they may be used to designate subprograms and also declared objects.

A new attribute designator, `Access`, has been defined for creating an access value designating a subprogram or object specified in the prefix. For example:

```
A := Object'Access;      -- point to a declared object
B := Subprogram'Access; -- point to a subprogram
```

Full type checking is performed as part of interpreting the `Access` attribute. An additional accessibility check is performed to ensure that the lifetime of the designated subprogram or object will not end before that of the access type, eliminating the possibility of dangling references.

Although these two extensions to access types share some common terminology and concepts the details are rather different and so we will now discuss them separately in the following sections.

3.7.1 Access to General Objects

Access types that may designate declared objects are called general access types, as distinguished from pool-specific access types, which correspond to those which were provided by Ada 83.

There are two steps to the use of general access types

- Objects that are to be designated by access values must be aliased. This can be done by using the reserved word **`aliased`** in their declaration. This serves various purposes. It documents the fact that an object is to be designated by an access value. It forces the object to be properly aligned in memory and informs the compiler that the representation of the object should correspond to that used for objects created by an allocator. In addition, the optimizer is informed that this object is likely to be accessible via one or more access values, and therefore its value might change as a result of an update via an access value.

- The attributes `Access` and `Unchecked_Access`, when applied to an object, return an access value that designates the object. Unless the subtype is tagged or has unconstrained discriminants, the subtype of the object must statically match the designated subtype of the access type. The access values which are formed by the `Access` attribute must obey certain accessibility restrictions, which are generally checked at compile time (at runtime in the case of access parameters). They cannot be used to create access values of a type whose lifetime is longer than the lifetime of the designated object; this prevents an access value from being stored in a global and then leaving the region where the designated object is declared. The access values that are formed by the `Unchecked_Access` attribute are not subject to such restrictions. It is the responsibility of the programmer who uses such unchecked access values to avoid dangling references.

General access types have the reserved word **`all`** or **`constant`** in their definition. We originally considered allowing any (object) access type to designate a declared object (as opposed to an allocated object), but this would have forced all access types to be represented as full addresses. By distinguishing general access types from pool-specific access types, we preserve the possibility of optimizing the representation of a pool-specific access type, by taking advantage of its limited storage-pool size.

A value of a general access type declared with the reserved word **`all`** can only designate variables (not constants), and may be used to read and update the designated object. If the reserved word **`constant`** is used, then access values may designate constants, as well as variables. An object designated by an access-to-constant value may not be updated via the access value. An allocator for an access-to-constant type requires an initial value and might generally reserve storage in a read-only part of the address space.

There are two important cases where a view is deemed to be aliased (and thus `Access` can be applied) even though the word **`aliased`** does not appear. One is that a parameter of a tagged type is considered to be aliased (see 6.1.2) and the other is where an inner component refers to the current instance of an outer limited type (see 4.6.3).

There is a restriction concerning discriminated records which ensures that we cannot apply the `Access` attribute to a component that might disappear. This is similar to the rule for renaming which prevents the renaming of a component of an unconstrained variable whose existence depends upon a discriminant.

Indirect access to declared objects is useful for avoiding dynamic allocation, while still allowing objects to be inserted into linked data structures. This is particularly useful for systems requiring link-time elaboration of large tables, which may use levels of indirection in their representation. Such access types are also convenient for returning a reference to a large global object from a function, allowing the object to be updated through the returned reference if desired.

Finally, rather than relying on allocators, it is sometimes appropriate to use a statically allocated array of objects, managed explicitly by the application. However, it may still be more convenient to reference components of the array using access values. By declaring the array components as aliased, the `Access` attribute may be used to produce an access value designating a particular component.

An interesting example is provided by the following which illustrates the static creation of ragged arrays

```
package Message_Services is
   type Message_Code_Type is range 0..100;

   subtype Message is String;

   function Get_Message(Message_Code: Message_Code_Type)
      return Message;

   pragma Inline(Get_Message);
end Message_Services;

package body Message_Services is
   type Message_Handle is access constant Message;

   Message_0: aliased constant Message := "OK";
   Message_1: aliased constant Message := "Up";
   Message_2: aliased constant Message := "Shutdown";
   Message_3: aliased constant Message := "Shutup";
   ...

   Message_Table: array (Message_Code_Type) of
      Message_Handle :=
         (0 => Message_0'Access,
          1 => Message_1'Access,
          2 => Message_2'Access,
          3 => Message_3'Access,
          -- etc.
         );

   function Get_Message(Message_Code: Message_Code_Type)
      return Message is
   begin
      return Message_Table(Message_Code).all;
   end Get_Message;
end Message_Services;
```

This example is based on Revision Request 018 and declares a static ragged array. The elements of the array point to strings, the lengths of which may differ. The access values are generated by the `Access` attribute; no dynamic allocation is needed to create the values.

Access types are used extensively in object-oriented applications. To enable the use of access types with the run-time dispatching provided for the primitive operations of tagged types, Ada 95 includes a new kind of **in** parameter, called an access parameter. We can thus write

```
procedure P(A: access T);
```

This is to be distinguished from a parameter of a named access type which already existed in Ada 83. A similar distinction arises with access discriminants as we saw in 3.4.1.

An access parameter is matched by an actual operand of any access type with the same designated type. Furthermore, if a subprogram has an access parameter with designated type `T`, and the subprogram is defined in the same package specification as the type `T`, then the subprogram is a primitive operation of `T`, and dispatches on the tag of the object designated by the access parameter. Inside the subprogram, an access parameter is of an anonymous general access type, and must either be dereferenced or explicitly converted on each use, or passed to another operation as an access parameter.

An important property of access parameters is that they can never have a null value. It is not permitted to pass null as an actual parameter (this is checked on the call) and of course being of an anonymous type another such object cannot be declared inside the subprogram. As a consequence within the subprogram there is no need to check for a null value of the type (neither in the program text nor in the compiled code). Note also that since other objects of the type cannot be declared, assignment and equality do not apply to access parameters.

For a tagged type `T` and an aliased object `X` of type `T`, `X'Access` and **`new`** `T` are overloaded on all access to `T`, on all access to `T'Class`, and on all other access to class-wide types that cover `T`. These overloadings on access to class-wide types allow allocators and the `Access` attribute to be used conveniently when calling class-wide operations, or building heterogeneous linked data structures.

Access parameters and access discriminants are important with respect to accessibility which we will now discuss in more detail. The accessibility rules ensure that a dangling reference can never arise; in general this is determined statically. Suppose we have a library package `P` containing a globally declared access type and a global variable

```
package P is
   type T is ...;
   type T_Ptr is access all T;
   Global: T_Ptr;
end P;
```

then we must ensure that the variable `Global` is never assigned an access to a variable that is local. So consider

```
procedure Q is
   X: aliased T;
   Local: T_Ptr := X'Access;    -- illegal
begin
   Global := X'Access;          -- illegal
   ...
   Global := Local;
   ...
end Q;
```

in which we have declared a local variable `X` and a local access variable. The assignment of `X'Access` to `Global` is clearly illegal since on leaving the procedure `Q`, this would result in `Global` referring to a non-existent variable. However, because we can freely assign access values, we must not assign `X'Access` to `Local` either since although that would be safe in the short term, nevertheless we could later assign `Local` to `Global` as shown.

Since we do not wish to impose accessibility checks at run-time on normal access assignment (this would be a heavy burden), we have to impose the restriction that the `Access` attribute can only be applied to objects with at least the lifetime of the access type. The rules that ensure this are phrased in terms of accessibility levels and the basic rule is that the access attribute can only be applied to an object at a level which is not deeper than that of the access type; this is, of course, known at compile time and so this basic accessibility rule is static. This may seem rather surprising since the concept of lifetime is dynamic and so one might expect the rules to be dynamic. However, it can be shown that in the case of named access types, the static rule is precisely equivalent to the intuitive dynamic rule. The reason for this is that the access attribute can only be applied at places where both the object and the access type are in scope; see [Barnes 95] for a detailed analysis. As discussed below, the situation is quite different for access parameters where the type has no name and the checks then have to be dynamic. (In the case of generic bodies, the rule is also dynamic as discussed in 12.3.)

Similar problems arise with discriminants and parameters of named access types. Thus we could not declare a local record with a component of the type `T_Ptr`. However, access discriminants and access parameters behave differently.

- The anonymous type is essentially declared inside the object or subprogram itself.

- It is not possible to have other components or objects of the same type (since it is anonymous) and they are treated as constants.

- Records with access discriminants have to be limited.

The net result is that the accessibility problems we encountered above do not arise. Revisiting the above example we can write

```
package P is
   type T is ...;
   type T_Ptr is access all T;
   type Rec(D: access T) is limited
      record
         ...
      end record;
   Global: T_Ptr;
   Global_Rec: Rec(...);
end P;
```

where we have added the record type `Rec` with an access discriminant `D` plus a global record variable of that type. Now consider

```
procedure Q is
   X: aliased T;
   Local_Rec: Rec(D => X'Access);      -- OK
begin
   Global := Local_Rec.D;         -- illegal, type mismatch
   Global := T_Ptr(Local_Rec.D); -- illegal, accessibility check
   Global_Rec := Local_Rec;      -- illegal, assignment limited type
   ...
end Q;
```

in which we have declared a local record variable with its access discriminant initialized to access the local variable `X`. This is now legal and the various attempts to assign the reference to `X` to a more global variable or component are thwarted for the various reasons shown. The straight assignment of the discriminant fails because of a type mismatch. The attempt to circumvent this problem by converting the access type also fails because of an accessibility check on conversions between access types [RM95 4.6]. And the attempt to assign the whole record fails because it is limited.

Access parameters are particularly important since they are the one case where an accessibility check is dynamic (other than in generic bodies). An access parameter carries with it an indication of the accessibility level of the actual parameter. Dynamic checks can then be made when necessary as for example when converting to an external named access type. Consider

```
procedure Main is
   type T is ...;
   type A is access all T;
   Ptr: A := null;
   procedure P(XP: access T) is
   begin
      Ptr := A(XP);      -- conversion with dynamic check
   end P;
   X: aliased T;
begin
   P(X'Access);
end Main;
```

The conversion compares the accessibility level of the object X passed as parameter with that of the destination type A; they are both the same and so the check passes. Observe that if the destination type A were declared inside P then the check can be (and is) performed statically. So not all conversions of access parameters require dynamic checks.

Another possibility is where one access parameter is passed on as an actual parameter to another access parameter. There are a number of different situations that can arise acording to the relative positions of the subprograms concerned; the various possibilities are analysed in detail in [Barnes 95] where it is shown that the implementation technique given in [AARM 3.10.2(22)] precisely meets the requirements of the rules. The rules themselves are in [RM95 3.10.2 and 4.6].

Without access parameters, the manipulation of access discriminants would be difficult. Given

```
procedure P(A: access T);
```

then we can satisfactorily make calls such as

```
P(Local_Rec.D);
```

in order to manipulate the data referenced by the discriminant. On the other hand declaring

```
procedure P(A: T_Ptr);
```

would be useless for the manipulation of the discriminant because the necessary type conversion on the call would inevitably be illegal for reasons of accessibility mentioned above.

As a first example of the use of access discriminants we will consider the case of an iterator over a set. This is typical of a situation where we want a reference from one object to another. The iterator contains a means of referring to the set in question and the element within it to be operated upon next. Consider

```
generic
   type Element is private;
package Sets is
   type Set is limited private;
   ... -- various set operations
   type Iterator(S: access Set) is limited private;
   procedure Start(I: Iterator);
   function Done(I: Iterator) return Boolean;
   procedure Next(I: in out Iterator);
   function Get_Element(I: Iterator) return Element;
   procedure Set_Element(I: in out Iterator; E: Element);
private
```

```
      type Node;
      type Ptr is access Node;
      type Node is
         record
            E: Element;
            Next: Ptr;
         end record;
      type Set is new Ptr;   -- implement as singly-linked list

      type Iterator(S: access Set) is
         record
            This: Ptr;
         end record;

   end Sets;

   package body Sets is
      ... -- bodies of the various set operations

      procedure Start(I: in out Iterator) is
      begin
         I.This := Ptr(I.S.all);
      end Start;

      function Done(I: Iterator) return Boolean is
      begin
         return I.This = null;
      end Done;

      procedure Next(I: in out Iterator) is
      begin
         I.This := I.This.Next;
      end Next;

      function Get_Element(I: Iterator) return Element is
      begin
         return I.This.E;
      end Get_Element;

      procedure Set_Element(I: in out Iterator; E: Element) is
      begin
         I.This.E := E;
      end Set_Element;

   end Sets;
```

The subprograms `Start`, `Next` and `Done` enable us to iterate over the elements of the set with the component `This` of the iterator object accessing the current element; the subprograms `Get_Element` and `Set_Element` provide access to the current element. The iterator could then be used to perform any operation on the values of the elements of the set.

As a trivial example the following child function `Sets.Count` simply counts the number of elements in the set. (Incidentally note that the child has to be generic because its parent is generic.)

```
   generic
   function Sets.Count(S: access Set) return Natural;
      -- Return the number of elements of S.

   function Sets.Count(S: access Set) return Natural is
```

```
      I: Iterator(S);
      Result: Natural := 0;
   begin
      Start(I);
      while not Done(I) loop
         Result := Result + 1;
         Next(I);
      end loop;
      return Result;
   end Sets.Count;
```

In the more general case the loop might be

```
   Start(I);
   while not Done(I) loop
      declare
         E: Element := Get_Element(I);   -- get old value
      begin
         ...                             -- do something with it
         Set_Element(I, E);              -- put new value back
         Next(I);
      end;
   end loop;
```

Note that if `Iterator.S` were a normal component rather than an access discriminant then we would not be able to initialize it at its point of declaration and moreover we could not make it point to `Sets.Count.S` without using `Unchecked_Access`.

Finally note that the procedure `Start` could be eliminated by declaring the type `Iterator` as

```
type Iterator(S: access Set) is
   record
      This: Ptr := Ptr(S.all);
   end record;
```

and this would have the advantage of preventing errors caused by forgetting to call `Start`.

3.7.2 Access to Subprograms

Ada 95 provides access-to-subprogram types. A value of such a type can designate any subprogram matching the profile in the type declaration, whose lifetime does not end before that of the access type. By providing access-to-subprogram types, Ada 95 provides efficient means to

- dynamically select and invoke a subprogram with appropriate arguments,
- store references to subprograms in data structures,
- parameterize subprograms with other subprograms (at run-time).

Access-to-subprogram values are created by the `Access` attribute. Compile-time accessibility rules ensure that a subprogram designated by an access value cannot be called after its enclosing scope has exited. This ensures that up-level references from within the subprogram will be meaningful when the subprogram is ultimately called via the access value. It also allows implementations to create and dereference these access-to-subprogram values very efficiently, since they can be a single address, or an address plus a "static link".

For Subprogram'Access, the designated subprogram must have formal parameter and result subtypes and a calling convention that statically match those of the access type. This allows the compiler to emit the correct constraint checks, and use the correct parameter passing conventions when calling via an access-to-subprogram value, without knowing statically which subprogram is being called. We call this subtype conformance.

Overload resolution of the Access attribute applied to an overloaded subprogram name represents a new situation in Ada. In Ada 83, the prefix of an attribute was required to be resolvable without context. However, for the Access attribute to be useful on overloaded subprograms, it was necessary to allow the Access attribute to use context to resolve the prefix. Therefore, if the prefix of Access is overloaded, then context is used to determine the specific access-to-subprogram type, and then the parameter and result type profile associated with that access type is used to resolve the prefix.

Indirect access to a subprogram is extremely useful for table-driven programming, using, for example, a state machine model. It is also useful for installing call-backs in a separate subsystem (like the X window system). Finally, it often provides an alternative to generic instantiation, allowing a non-generic parameter to be a pointer to a subprogram, such as for applying an operation to every element of a list, or integrating a function using a numerical integration algorithm.

A number of examples of the use of access to subprogram types will be found in II.5. However a very important use is to provide much better ways of interfacing to programs written in other languages. This is done in conjunction with the pragma Import (essentially replacing Interface) and new pragmas Export and Convention. For details see Part Three.

It should be noted that there is no equivalent to access discriminants or access parameters for access to subprogram types. Apart from any aesthetic consideration of writing such an in situ definition, the key reason concerns the implementation problems associated with keeping track of the environment of such a "subprogram value". As a consequence we cannot, for example, use the access to a local procedure as a parameter of a more globally declared procedure. Such values would in any case not be safely assignable into a global.

The accessibility restrictions mean that access to subprogram values do not provide a mechanism to solve the general iterator problem where the essence is usually to apply some inner procedure over every element of a set with the inner procedure having access to more global variables. One alternative approach is to use access discriminants as discussed in 3.7.1; another, perhaps better, approach is to use type extension as illustrated in 4.4.4.

Generic formal subprograms remain the most general means of parameterizing an algorithm by an arbitrary externally specified subprogram. Moreover they are often necessary anyway. For example, consider a typical mathematical problem such as integration briefly mentioned in II.5. In practice the integration function would inevitably be generic with respect to the floating point type. So a more realistic specification would be

```
generic
   type Float_Type is digits <>;
package Generic_Integration is
   type Integrand is
                    access function(X: Float_Type) return Float_Type;

   function Integrate(F: Integrand; From, To: Float_Type;
                Accuracy: Float_Type := 10.0*Float_Type'Model_Epsilon)
                                                      return Float_Type;
end Generic_Integration;
```

Suppose now that we wish to integrate a function whose value depends upon non-local variables and that therefore has to be declared at an inner level. All that has to be done is to instantiate the generic at the same inner level and then no accessibility problems arise. So

```
with Generic_Integration;
procedure Try_Estimate(External_Data: Data_Type;
                       Lower, Upper: Float;
                             Answer: out Float) is
   -- external data set by other means

   function Residue(X: Float) return Float is
      Result: Float;
   begin
      -- compute function value dependent upon external data
      return Result;
   end Residue;

   package Float_Integration is
         new Generic_Integration(Float_Type => Float);
   use Float_Integration;

begin
   ...
   Answer := Integrate(Residue'Access, Lower, Upper);

end Try_Estimate;
```

The key point is that the instantiated access type `Integrand` is at the same level as the local function `Residue` and therefore the `Access` attribute can be applied. This technique can of course be used even when there are no generic parameters.

3.8 Type Conversion

Because Ada 95 supports type extension and has more flexible access types, the possibilities and needs for type conversion become much more extensive than in Ada 83. In Ada 83, type conversion involved only a possible representation change and a possible constraint check. If the conversion succeeded, no components were lost or added, and the conversion was always reversible. There were only three kinds of conversions, between derived types, between numeric types and between array types. Note in particular that there were no conversions between access types (except for the case where the type itself was derived from another access type).

Conversions in Ada 95 are classified as view conversions and value conversions. The general idea is that a view conversion doesn't really perform a conversion but just provides a different view of the object.

View conversions arise in two situations, where the operand is an object of a tagged type, and where the conversion is used as an actual parameter corresponding to a formal in out or out parameter. Other conversions are value conversions. Another way of looking at the difference is that view conversions are for situations where an object is being converted whereas a value conversion can apply to an expression.

The use of a view conversion as a parameter existed in Ada 83; for example where a conversion of an object of say type `Integer` was used as an actual parameter corresponding to a formal in out parameter of type `Float`. Such view conversions cause a real change of representation in both directions and indeed view conversions of nontagged types are always reversible.

View conversions of tagged types are different; no change of representation ever occurs; we merely get a different view seeing different properties of the same object. And view conversions of tagged types are generally not reversible because of the possibility of type extension.

For tagged specific types there is an important rule that conversion can only be towards the root type. Conversion of a specific type away from the root type is not possible because additional components will generally be required. Such additional components can be provided by an extension aggregate. As we saw in the example of type `Object` and its extension `Circle` from II.1 we can write

```
Object(C)
```

as an acceptable conversion towards the root but must write

```
C: Circle;
O: Object;
...
C := (O with Radius => 12.0);
```

to perform the operation away from the root. (In an early draft of Ada 9X a form of conversion was used for such an extension but it was felt to be confusing and overcomplicate the rules for conversion; it also had generic contract problems.) Note that we have used the named notation for the additional components (in this case only one). Another important point is that the ancestor expression (before **with**) need not be the immediate ancestor of the target type but can be any ancestor. See also 3.6.1.

The same principle applies in the case of conversions from a tagged class-wide type to a specific type; conversion is only allowed if the tag of the current value of the class-wide object is such that conversion is not away from the root. This is not known statically and so a tag check verifies that the type identified by the tag of the operand matches that of the target type, or is a derivative of it. `Constraint_Error` is raised if this check fails.

Conversion from a specific type to a class wide type is always allowed implicitly (that is no conversion need be explicitly stated); of course the specific type must be in the class concerned. We could not convert a `Low_Alert` to `Medium_Alert'Class`; any attempt would be detected at compile time.

Conversion from one class-wide type to another is also possible. The classes obviously have to have a common ancestor and it may be necessary to check the tag at runtime. If the source class is the same as or a subclass of the target class then clearly no check is necessary.

We will consider the conversion of tagged types in more detail when we discuss redispatching in the next chapter (see 4.5).

Conversion between access types was not possible in Ada 83 (unless one was derived from another); each access type was considered unrelated (even if the accessed types were the same). Another issue was that access values need not necessarily be held as addresses but could be indexes into the relevant pool.

However, the introduction of general access types and access parameters means that the conversion between access types is very necessary.

Conversion from pool specific types to general access types and between general access types is therefore permitted provided the accessed types are the same or are suitably related. But we cannot convert from an access to constant type to an access to variable type because we might thereby obtain write access to a constant. In general a conversion may involve constraint and accessibility checks.

Conversions are particularly useful for programming with access types designating tagged types. Essentially, an access type conversion is permitted if access values of the target type may "safely" designate the object designated by the operand access value; or in other words providing the new view is acceptable for the designated object. Thus a conversion from an access to class-wide type to an access to specific type will require a dynamic check to ensure that the designated object is of the specific type (or derived from it). Generally, conversions between access types to tagged types follow exactly the same rules and involve the same checks as conversions between the designated types. Both conversions effectively give new views of the object concerned.

Conversions between access types (in general) may require accessibility checks to ensure that the new value could not give rise to a dangling reference. It is possible to convert between any general access types (including anonymous access types used as access parameters and discriminants) provided the designated types are the same. An example of access type conversion where an accessibility check is required occurs in 3.7.1. Conversions between access types may also require constraint checks to ensure that any constraints on the accessed subtype are satisfied.

As explained above we have generalized implicit subtype conversions on arrays ("sliding") to apply in more circumstances. These new rules should minimize the times when an unexpected `Constraint_Error` arises when the length of the array value is appropriate, but the upper and lower bounds do not match the applicable index constraint. In effect, we are treating the bounds as properties of array objects rather than of array values. Array values have a length for each dimension, but the bounds may be freely readjusted to fit the context.

Note also that array conversions require that the component subtypes statically match in Ada 95 whereas the check was dynamic in Ada 83. This is a minor incompatibility but avoids unnecessary runtime checks.

3.9 Staticness

In Ada 83, static expressions were limited to predefined operators applied to static operands, to static attributes or to static qualified expressions. For Ada 95, we have extended the rules so that a static expression can also include items such as membership tests and attributes with static constituents. See [RM95 4.9] for details.

By allowing more constructs in static expressions, the programmer has more freedom in contexts where static values are required. In addition, we ensure more uniformity in what expressions are evaluated at compile-time. Some Ada 83 compilers were aggressive in evaluating compile-time known expressions, while others only evaluated those expressions that were "officially" static. By shifting the definition of static to more closely correspond to compile-time-known, uniformity of efficiency is enhanced.

In addition to generalizing the rules for static expressions, we also require that all static evaluation be performed exactly. Although many compilers already perform all compile-time arithmetic with arbitrary precision, this rule will provide more predictability for the value of a static expression. Note that the exact static value must still be converted to a machine manipulable representation when combined in an expression with non-static values.

Static strings are also introduced for use as parameters of pragmas. There are no other contexts in the standard which require static strings.

Staticness is also relevant in other situations such as subtype conformance (see 6.2). This kind of conformance is required between the parameter and result specifications given in an access to subprogram type definition, and the specification of a potential designated subprogram. Subtype conformance is based on a static match between the subtypes of corresponding parameters and the result, if any. This is necessary because when calling a subprogram via an access to subprogram type, the actual parameters must be prepared and the call must be performed given only the address (and perhaps a static link) for the target subprogram. The way parameters are passed, the constraints that need to be checked, the way the result is returned, and any other calling conventions must be determined completely knowing only the definition of the access-to-subprogram type.

A static subtype match is required for access-to-subprogram matching so that no additional checks on the actual parameters are required when calling indirectly through an access to subprogram type.

There is a general philosophy that static matching is required when two subtypes are involved whereas when only one subtype and a value are involved (as in assignment) then dynamic matching (possibly resulting in `Constraint_Error`) is applied as it was in Ada 83. Thus static matching is also required in matching the component subtypes in array conversions and matching the subtype indication in deferred constants. This change of philosophy eliminates a number of run-time checks and makes for the earlier detection of errors in such situations.

3.10 Other Improvements

Ada 83 had a restriction that type, subtype, and object declarations were not permitted after bodies (including body stubs) within a declarative part. This restriction has been removed in Ada 95. The original restriction reflected Ada's Pascal heritage, where the ordering restrictions between declarations are even more restrictive. However, in retrospect, the restriction seems somewhat arbitrary, and forces the separation of declarative items that might more naturally be grouped together, particularly in a package body declarative part.

By removing this restriction, it becomes legal to move local variable declarations of a subprogram body to the end of its declarative part. This ensures that such variables are not accessible for up-level references from nested subprograms declared in the same declarative part. By so doing, it makes it easier for a compiler to allocate such variables to hardware registers, rather than having to keep them in memory locations to support possible up-level references.

Having removed this restriction, it is necessary to rely more heavily on the Ada 83 rule [RM83 13.1(5-7)] that the representation for a type is "frozen" by the appearance of a body (including a body stub). This rule precludes the separation of a representation clause from its associated declaration by a body. In Ada 83, this requirement was a ramification of the syntax, since representation clauses were not allowed to follow bodies syntactically. In Ada 95, the requirement becomes more relevant, since representation clauses are syntactically allowed to appear anywhere in a declarative part.

3.11 Requirements Summary

The requirements for international users

R3.1-A(1) — Base Character Set

R3.1-A(2) — Extended Graphic Literals

R3.1-A(3) — Extended Character Set Support

are met by the changes to the type `Character` and the introduction of `Wide_Character` and associated packages as discussed in 3.2.

The study topic and two requirements regarding subprograms

S4.1-A(1) — Subprograms as Objects

R4.1-B(1) — Passing Subprograms as Parameters

R4.1-B(2) — Pragma Interface

are met by access to subprogram types and the pragmas `Import`, `Export` and `Convention` as discussed in 3.7.2.

The requirement

R6.1-A(1) — Unsigned Integer Operations

is met by the modular types described in 3.3.2.

The requirement

R6.4-A(1) — Access Values Designating Global Objects

is met by general access types and the study topic

S6.4-B(1) — Low-Level Pointer Operations

is also addressed by general access types and the attribute `Unchecked_Access`.

The requirement

R10.1-A(1) — Decimal-Based Types

is met by the decimal types mentioned in 3.3.3. Full support for decimal types is provided by the Information Systems annex to which the reader is referred for further details.

The requirement

S2.3-A(1) — Improve Early Detection of Errors

is addressed by the introduction of static subtype matching.

The requirement

R2.2-A(1) — Reduce Deterrents to Efficiency

is addressed by the introduction of the concept of base range for numeric types as discussed in 3.3 and by the removal of the restriction on the order of declarations mentioned in 3.10.

The requirement

R2.4-A(1) — Minimize Implementation Dependencies

is addressed by the stipulation that all static expressions are evaluated exactly and that rounding of odd halves is always away from zero; see 3.9 and II.12.

Finally we have mentioned one of the items listed under the general requirement

R2.2-B(1) — Understandability

which is that the restriction on order of declarations is now removed.

4 Object Oriented Programming

This chapter describes the various ways in which object oriented programming is achieved in Ada 95. The main facilities upon which this is based are

- Record types which are marked as tagged may be extended with additional components on derivation.
- The `Class` attribute may be applied to a tagged type and denotes the corresponding class-wide type.
- Subprogram calls with formal parameters of a specific type called with actual parameters of a class-wide type are dispatching calls.
- Types and subprograms may be specified as abstract.
- There are various new forms of generic parameter corresponding to derived and tagged types.

These topics were discussed in some detail in Part One and are further discussed in other chapters in this part (see Chapter 3 for types, including abstract types and abstract subprograms, and Chapter 12 for generic parameters). The discussion in this chapter is more idiomatic and concentrates on how the features are used in Ada 95 and briefly contrasts the approach with that of other object oriented languages.

4.1 Background and Concepts

Ada has been traditionally associated with object oriented design [Booch 86], which advocates the design of systems in terms of abstract data types using objects, operations on objects, and their encapsulation as private types within packages. The "ingredients" of object oriented design may be summarized as follows:

Objects. Entities that have structure and state.

Operations. Actions on objects that may access or manipulate that state.

Encapsulation. Some means of defining objects and their operations and providing an abstract interface to them, while hiding their implementation details.

Ada 83 was well suited to supporting the paradigm of object oriented design. Object oriented programming, as that term has evolved over the past decade, builds upon the base of object oriented design, adding two other ingredients: inheritance and polymorphism. While the specific properties of these two facilities vary from one programming language to another, their essential characteristics may be stated as

Inheritance. A means for incrementally building new abstractions from existing ones by "inheriting" their properties — without disturbing the implementation of the original abstraction or the existing clients.

Polymorphism. A means of factoring out the differences among a collection of abstractions, such that programs may be written in terms of their common properties.

Ada 83 has been described as an object based language; it does not have the support for inheritance and polymorphism found in fully object oriented languages (see 4.7). Recognizing this, the Ada 9X Requirements reflect the need to provide improved support for this paradigm through three Study Topics [DoD 90] as follows.

> S4.1-A(1) — Subprograms as Objects: Ada 9X should provide:
>
> 1 an easily implemented and efficient mechanism for dynamically selecting a subprogram that is to be called with a particular argument list;
>
> 2 a means of separating the set of subprograms that can be selected dynamically from the code that makes the selection.
>
> S4.3-A(1) — Reducing the Need for Recompilation: Ada 9X recompilation and related rules should be revised so it is easier for implementations to minimize the need for recompilation and for programs to use program structures that reduce the need for recompilation.
>
> S4.3-B(1) — Programming by Specialization/Extension: Ada 9X shall make it possible to define new declared entities whose properties are adapted from those of existing entities by the addition or modification of properties or operations in such a way that:
>
> - the original entity's definition and implementation are not modified;
>
> - the new entity (or instances thereof) can be used anywhere the original one could be, in exactly the same way.

Each of these Study topics can be understood in relation to object oriented programming. S4.1-A(1) seeks the ability to associate operations (subprograms) with objects, and to dynamically select and execute those operations. This is one basis on which to develop run-time polymorphism.

Among the various causes of excessive recompilation addressed by S4.3-A(1) are those arising from the breakage of an existing abstraction for the purpose of extending or otherwise reusing it to build a new abstraction.

The topic S4.3-B(1) implies the essence of object oriented programming as defined above. Alternatively, one might think of this in terms of two programming paradigms

Variant programming. New abstractions may be constructed from existing ones such that the programmer need only specify the differences between the new and old abstractions.

Class-wide programming. Classes of related abstractions may be handled in a unified fashion, such that the programmer may systematically ignore their differences when appropriate.

Finally, it should be mentioned that there are two rather awkward problems to be addressed and solved in designing a compiled language for object oriented programming which retains efficiency and avoids unnecessary run-time decisions.

Dispatching. The means of indicating in the program text when dispatching and especially redispatching is used as opposed to static resolution.

Multiple Inheritance. The means of inheriting components and operations from two or more parent types.

As will be seen, the solution adopted to these problems in Ada 95 illustrates our concern for clarity and the advantages of the building block approach.

4.2 General Approach

Ada 95 generalizes the type facilities of Ada 83 in order to provide more powerful mechanisms for variant and class-wide program development and composition. Derived types in Ada 83 provided a simple inheritance mechanism: they inherited exactly the structure, operations, and values of their parent type. This "inheritance" could be augmented with additional operations but not with additional components. Ada 95 generalizes type derivation to permit type extension as we saw in II.1.

A tagged record or private type may be extended with additional components on derivation. Tagged objects are self-identifying; the tag indicates their specific type. Tagged types provide a mechanism for single inheritance as found in object oriented programming languages such as Simula [Birtwistle 73] and Smalltalk [Goldberg 83].

The following example of type extension is inspired by [Seidewitz 91]. We first declare

```
type Account_With_Interest is tagged
   record
      Identity: Account_Number := None;
      Balance : Money := 0.00;
      Rate    : Interest_Rate := 0.05;
      Interest: Money := 0.00;
   end record;

procedure Accrue_Interest(On_Account: in out Account_With_Interest;
                          Over_Time : in Integer);

procedure Deduct_Charges(From: in out Account_With_Interest);
```

and can then extend it

```
type Free_Checking_Account is new Account_With_Interest with
   record
      Minimum_Balance: Money := 500.00;
      Transactions   : Natural := 0;
   end record;

procedure Deposit(Into  : in out Free_Checking_Account;
                  Amount: in Money);

procedure Withdraw(From  : in out Free_Checking_Account;
                   Amount: in Money);

Insufficient_Funds: exception;   -- raised by Withdraw

procedure Deduct_Charges(From: in out Free_Checking_Account);
```

The type `Account_With_Interest` is a tagged type. The type `Free_Checking_Account` is derived from it, inheriting copies of its components (`Identity`, `Balance`, `Rate`, `Interest`) and its operations (Accrue_Interest and Deduct_Charges). The derived type declaration has a record extension part that adds two additional components (Minimum_Balance and Transactions) to those inherited from the parent. The type adds some new operations (Deposit and Withdraw) and also overrides Deduct_Charges such that if the Balance was above the Minimum_Balance, no charges would be deducted. All components of the type, whether inherited or declared as a part of the extension, are equally accessible (unlike "nested" record types).

In Ada 83, the types declared in the visible part of a package had special significance for Ada's abstraction mechanisms. Such operations on user-defined types were first-class in a manner that parallels those of the predefined types. For derived types, these operations, together with the implicitly declared basic operations, were the derivable operations on a type.

With the increased importance of derived types for object oriented programming in Ada 95, the notion of the operations closely related to a type in this manner is generalized. The primitive operations of a type are those that are implicitly provided for the type and, for types immediately declared in a package specification, all subprograms with an operand or result of the type declared anywhere in that package specification. The domain is therefore extended to include the private part (but not the body).

Thus, in Ada 95, the derivable operations of Ada 83 have become "primitive operations" and the restriction of these operations to the visible part of a package has been eliminated. These changes support added capability: primitive operations may be private and a type and its derivatives may be declared in the same declarative region (this property is useful for building related abstractions and was used in the package New_Alert_System of Part One).

Primitive operations clarify the notion of an abstract data type for purposes of object oriented programming (inheritance and polymorphism) and genericity. They are distinguished from the other operations of a type in the following ways

Inheritance. Primitive operations are the derivable (inherited) operations.

Polymorphism. Primitive operations are dispatching operations on tagged types.

Genericity. Primitive operations are the ones available within generic templates parameterized by a class.

Ada 83 used the term "class" (see [RM83 3.3]) to characterize collections of related types. The class of a type determines how the type is declared, the types it can be converted to, its predefined operations, and its structure. The class of a generic formal type parameter determines the operations that are available within the generic template. Types within a class have common structure and operations (see III.1.2 for a further description of the class structure of Ada).

Ada 95 formalizes the Ada 83 notion of class. A type and its direct and indirect derivatives, whether or not extended, constitute a derivation class. This definition allows for user-defined classes based on derivation. User-defined classes, like the language-defined classes, support type conversion, may be used to parameterize generic units and, in the case of tagged types, provide class-wide programming.

Explicit conversion is defined among types within a class, as it was in Ada 83 for types related by derivation, except that conversion is not allowed away from the root since additional components may be required. Such transformations require an extension aggregate as described in 3.8.

Thus, continuing the previous example, a value of Account_With_Interest can be extended to a value of Free_Checking_Account by providing values for the additional components Minimum_Balance and Transactions.

```
Old_Account: Account_With_Interest;
...
New_Account: Free_Checking_Account :=
      (Old_Account with Minimum_Balance => 0.00, Transactions => 0);
```

The Ada 95 rules for conversion between types in a class define the semantics of inherited operations in Ada 95 and are consistent with the semantics of inherited operations in Ada 83. Calling an inherited operation is equivalent to calling the parent's corresponding operation with a conversion of the actual to the parent type. Thus, inherited operations "ignore" the extension part.

User-defined classes may be employed to parameterize generic units. A new kind of generic formal, a generic formal derived type, may be used. This kind of formal is matched by any type in the class rooted at the generic formal's specified ancestor type.

For each tagged type `T`, there is an associated class-wide type `T'Class`. The set of values of `T'Class` is the discriminated union of the sets of values of `T` and all types derived directly or indirectly from `T`. Discrimination between the different specific types is with a type tag. This tag, associated with each value of a class-wide type, is the basis for run-time polymorphism in Ada 95. Note that ordinary types are referred to as specific types to distinguish them from class-wide types.

The associated class-wide type `T'Class` is "dynamic" in the sense of [Abadi 91]. The values of the class-wide type can be thought of as pairs consisting of

- A tag. A type descriptor ranging over the types that are members of the class; and

- A value. The value taken from the specific type with the given tag.

Such tag and value pairs are strongly typed, consistent with the philosophy of Ada. But only the class, and not necessarily the type within that class, will generally be known statically.

Class-wide types have no primitive operations of their own. However, explicit operations may be declared for such types, using `T'Class` as a subtype mark. Such operations are "class-wide" and can be applied to objects of any specific type within the class as well as to objects of the class-wide type or a descendent class-wide type.

Thus the following functions

```
function Size_In_Bytes(Any_File: File'Class) return Natural;
function Get_File_From_User return File'Class;
```

are class-wide operations and can be applied to all specific types of the class of types derived from the tagged type `File`. No dispatching is involved; the one same function is called whatever the tag of the actual parameter.

On the other hand, when a primitive operation of a tagged type is called with an operand of the class-wide type, the operation to be executed is selected at run time based on the type tag of the operand. As mentioned before, this run-time selection is called dispatching, so primitive operations of tagged types are called dispatching operations. Dispatching provides a natural form of run-time polymorphism within classes of related (derived) types. This variety of polymorphism is known as "inclusion polymorphism" [Cardelli 85].

	formal	
actual	specific	class-wide
specific	static binding	class-wide op
class-wide	dispatching	class-wide op

Table 4-1: Kinds of Binding

An operand used to control dispatching is called a controlling operand. A primitive operation may have several controlling operands; a primitive function may also have a controlling result. For a further discussion on controlling operands and results see 4.5.

The different kinds of binding corresponding to the various combinations of actual and formal parameters are summarized in Table 4-1.

The following example shows how a type `File` might be the basis for a class of types relating to the implementation of an Ada library

```
type File is tagged private;
procedure View(F: File);
   -- display file F on screen

type Directory is new File with private;
procedure View(D: Directory);
   -- list directory D

type Ada_File is new File with private;
procedure View(A: Ada_File);
   -- open A with Ada sensitive editor

type Ada_Library is
   new Directory with private;
procedure View(L: Ada_Library);
   -- list library units of L and their status

declare
   A_File: File'Class := Get_File_From_User;
begin
   View(A_File);   -- dispatches according to specific type of file
end;
```

The above example presents a user-defined class of `File` types. The type `File` is tagged and hence the primitive operation `View` is dispatching. `View` is overridden for each type in the class in order to provide a unique behavior for each type of `File`. When `View` is called with a parameter that is of type `File'Class`, the tag will be used to determine the actual type within the class, and the call will dispatch to the `View` procedure for that type. On the other hand if `View` is called with a specific type then the choice of `View` procedure to be called is determined at compile time.

The hierarchy of types in the above example is illustrated in Figure 4-1.

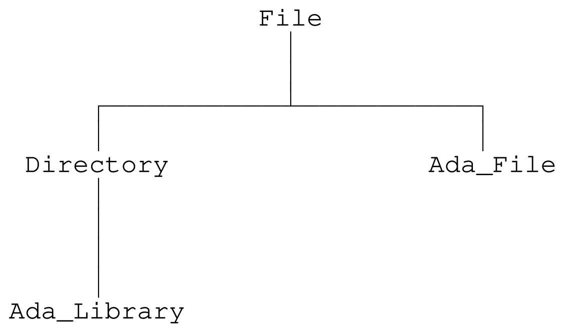

Figure 4-1: The File Hierarchy

An earlier version of Ada 9X introduced class-wide types through the `Class` attribute for all derivation classes and not just those for tagged types. This was discarded since many reviewers felt that the added flexibility was unwise.

Note also that universal types (for numeric types) behave much as class-wide types although there are differences, see 3.3.

When building an abstraction that is to form the basis of a class of types, it is often convenient not to provide actual subprograms for the root type but just abstract subprograms which can be replaced when inherited. This is only allowed if the root type is declared as abstract; objects of an abstract type cannot exist. This technique enables common class-wide parts of a system to be written without being dependent on the properties of any specific type at all. Dispatching always works because it is known that there can never be any objects of the abstract type and so the abstract subprograms could never be called. This technique is illustrated in II.3.

4.2.1 Benefits of Approach

A number of important practical criteria of concern to both existing and new users were taken into account when designing the object oriented facilities of Ada 95.

Compatibility

Legal Ada 83 programs should remain legal in Ada 95. Ideally, existing Abstract Data Types (ADTs) should be reusable with newly developed ones — the new facilities should not be so radically different from mechanisms of Ada 83 that existing ADTs must be rebuilt before being reused.

Tagged type extension and class-wide types are built upon the Ada 83 model of derived types. Their use is optional, and their presence in the language does not affect Ada 83 programs. Existing ADTs may be combined in some ways with new object oriented abstractions without modification. In other cases, it may be sufficient to add "tagged" to a type declaration, or to make other simple modifications such as changing an access type declaration to designate a class-wide type. Of course, in order to exploit these facilities to the full, it will be necessary to take them into account during the design process.

Consistency

The solution should be conceptually consistent with existing Ada programming models. Intuitions about objects, types, subprograms, generic units, and so on should be preserved.

Ada 95 provides new capabilities in the context of a unified programming model: including types, operations and generic units. Class-wide programming generalizes the classes developed in Ada 83 to user-defined classes.

Efficiency

The solution should offer efficient performance for users of the facility with, ideally, no distributed overhead for non-users.

The introduction of tagged types, and a distinct class-wide type associated with each specific type as the mechanism for dispatch, makes run-time polymorphism optional to programmers (in contrast to languages like Smalltalk), in two senses.

- Programmers can choose whether or not to use object oriented programming, by employing tagged and class-wide types. Types without tags incur no space or time overhead. Only class-wide types allow for class-wide type matching.

- Dispatching occurs only on primitive operations of tagged types and only when an actual operand is of a class-wide type.

Implementability

The solution should be readily implementable within current compiler technology, and provide opportunities for optimizations.

Dispatching may be implemented as an indirect jump through a table of subprograms indexed by the primitive operations. This compares favorably with method look-up in many object oriented languages, and with the alternative of variant records and case statements, with their attendant variant checks, both in implementability and run-time efficiency.

4.3 Class Wide Types and Operations

We have seen that a record or private type marked as tagged may be extended on derivation with additional components. The run-time tag identifies information that allows class-wide operations on the class-wide type to allocate, copy, compare for equality, and perform any other primitive operations on objects of the class-wide type, in accordance with the requirements of the specific type identified by the tag.

The tag is thus important to the inner workings of type extension and class-wide types in Ada 95 and is brought to the fore by using the reserved word **`tagged`** in the declaration of the type. The concept of a tag is of course not new to programming languages but has a long precedent of use in Pascal (where it is used in the sense of discriminant in Ada) and is discussed by Hoare in [Dahl 72]. More recently the phrase *type tag* has been used by Wirth in connection with type extension [Wirth 88].

The reader might find it helpful to understand the concept of the tag and the dispatching rules by considering the implementation model alluded to at the end of the last section. We emphasize that this is just a *possible* model and does not imply that an implementation has to be done this way. In this model the tag is a pointer to a dispatch table. Each entry in the table points to the body of the subprogram for a primitive operation. Dispatching is performed as an indirect jump through the table using the primitive operation as an index into the table.

As an illustration consider the class of `Alert` types declared in II.1 in the package `New_Alert_System`. These types form a tree as illustrated in II.2. Recall that the root type `Alert` has primitive operations, `Display`, `Handle` and `Log`. These are inherited by `Low_Alert` without any changes. `Medium_Alert` inherits them from `Alert` but overrides `Handle`. `High_Alert` inherits from `Medium_Alert` and again overrides `Handle` and also adds `Set_Alarm`. (For simplicity, we will ignore other predefined operations which also have "slots" in the dispatch table; such as assignment, the equality operator and the application of the `Size` attribute.)

The tags for the various types are illustrated in Figure 4-2. A dispatching call, such as to `Handle`, is very cheaply implemented. The code simply jumps indirectly to the contents of the table indexed by the fixed offset corresponding to `Handle` (one word in this example). The base of the table is simply the value of the tag and this is part of the value of the class-wide object. Note moreover that the dispatch table does not contain any class-wide operations (such as `Process_Alerts` in II.2) since these are not dispatching operations.

In addition to being used as formal parameters to class-wide operations, class-wide types may also be used as the designated type for an access type, and as the type of a declared object. Access types designating a class-wide type are very important, since they allow the creation of heterogeneous linked data structures, such as trees and queues.

```
         Alert'Tag   -->   Display     --> Display of Alert
                           ------------
                           Handle      --> Handle of Alert
                           ------------
                           Log         --> Log of Alert

     Low_Alert'Tag   -->   Display     --> Display of Alert
                           ------------
                           Handle      --> Handle of Alert
                           ------------
                           Log         --> Log of Alert

  Medium_Alert'Tag   -->   Display     --> Display of Alert
                           ------------
                           Handle      --> Handle of Medium_Alert
                           ------------
                           Log         --> Log of Alert

    High_Alert'Tag   -->   Display     --> Display of Alert
                           ------------
                           Handle      --> Handle of High_Alert
                           ------------
                           Log         --> Log of Alert
                           ------------
                           Set_Alarm   --> Set_Alarm of High_Alert
```

Figure 4-2: Tags and Dispatch Tables

Declared objects of a class-wide type are not as frequently used as are formal parameters and heap objects, but they are useful as intermediates in larger computations. However, because there is no upper bound on the size of types in a class, a declared object of a class-wide type must be explicitly initialized. This determines the size, the tag, and any discriminants for the object, and thereafter neither the tag nor the discriminants may be changed. Of course, it is necessary for all class-wide objects to have a tag so that dispatching works without any unnecessary tests. We have chosen to specify the tag by requiring an explicit initial value.

This indirectly provides a capability somewhat akin to declaration by association using **`like`** in Eiffel [Meyer 88]. We can thereby ensure, for example, that a locally declared class-wide object has the same tag as an actual parameter.

Note that discriminants of tagged types are not permitted to have defaults; this would have increased the complexity of the language to no great benefit since the tag of an object (specific and class-wide) cannot be changed and the tag is treated as a (hidden) discriminant. It would be inconsistent to allow discriminants to be changed but not tags.

If assignment were allowed to change the tag or the discriminants, then the size of the class-wide object might have to grow, requiring a deallocation and reallocation as part of assignment.

We have avoided introducing operations in Ada 95 that involve this kind of implicit dynamic allocation at run-time. Therefore, an explicit access value with explicit deallocation and reallocation is required if a programmer desires to have the equivalent of an unconstrained object of a class-wide type.

Note that Ada 83 required a similar approach for handling unconstrained arrays and unconstrained discriminated types without defaults for the discriminants. In general, the way unconstrained composite types and their associated bounds or discriminants were handled in Ada 83 is a good model for how class-wide types and their associated type tags are handled in Ada 95.

The predefined equality operators and the membership tests are generalized to apply to class-wide types. Like other predefined operations on such types, the implementation will depend on the particular specific type of the operands. Unlike normal dispatching operations, however, `Constraint_Error` is not raised if the tags of the operands do not match.

For equality, tag mismatch is treated as inequality. Only if the tags match is a dispatch then performed to the type-specific equality checking operation. This approach allows a program to safely compare two values of a class-wide tagged type for equality, without first checking that their tags match. The fact that no exception is raised in such an equality check is consistent with the other predefined relational operators, as noted in [RM83 4.5.2(12)].

For a membership test such as `X in S`, where `X` is of a class-wide type, the tag of the value of the simple expression `X` is checked to see whether it belongs to the subtype mark `S`. If the subtype mark `S` identifies a class-wide type, then the membership test determines whether the tag of the value identifies a specific type covered by the class-wide type. If the subtype mark `S` identifies a specific tagged type, then the membership test determines whether the tag of the value equals the tag of that type. In any case, to be considered a member, the value must satisfy any constraints associated with the subtype mark.

The `Tag` attribute is defined for querying the tag of a specific type, or of an object of a class-wide type. This allows two class-wide objects to be checked to see whether they have the same tag. It is also possible to compare the tag of a class-wide object with the tag of a specific type; such a comparison is equivalent to a membership test.

Thus (using the alert example from Part One), the test

```
AC in Medium_Alert
```

is identical to

```
AC'Tag = Medium_Alert'Tag
```

but note that the test

```
AC in Medium_Alert'Class
```

has no generally applicable equivalent in terms of explicit user checks on tags because we cannot talk about possible future extensions in terms of tags. It is thus preferable to use membership tests rather than explicit testing of tags in order to ensure that our program is extensible.

The rules for membership tests on class-wide types are constructed so that certain simple type-specific behavior may be performed in a class-wide operation, without the need to declare and define a new primitive operation on all types within the class.

For example, given

```
type Expression_Node is tagged...

type Binary_Operator is new Expression_Node with ...

type Node_Ptr is access Expression_Node'Class;
```

one could define the following operation `Display` on the access to class-wide type `Node_Ptr`

```
procedure Display(Expr: Node_Ptr; Prec: Positive := 1) is
 -- display expr, parenthesized if necessary
begin
   if Expr.all in Binary_Operator'Class then
      -- handle the binary operator subclass
      declare
         Binop: constant Bin_Op_Ptr := Bin_Op_Ptr(Expr);
            -- convert parameter to ptr to Binary_Operator
            -- to gain access to its subclass operations
      begin
         if Precedence(Binop) < Prec then
            -- parenthesize if lower precedence
            Put('(');
         end if;
         -- display left, op, right, passing down precedence
         Display(Binop.Left, Precedence(Binop));
         Put(Symbol(Binop));
         Display(Binop.Right, Precedence(Binop));
         if Precedence(Binop) < Prec then
            -- closing parenthesis if necessary
            Put(')');
         end if;
      end;
   else
      -- handle the other kinds of expressions
      ...
   end if;
end Display;
```

An alternative, more "object-oriented" approach would be to define a separate `Display` primitive operation for each distinct type within the class. See for example [Taft 93].

We conclude this section by discussing a number of important general principles regarding primitive operations and dispatching. It is instructive to map these principles into the implementation model of the tag and dispatch table mentioned above; but remember that this is only a possible model although a very natural one. We can refer to the entries in the dispatch table as "slots".

The first general principle is that dispatching always works without any checking at runtime; in other words that the subprogram referred to by the dispatch table for the tag value can always be safely called. A number of individual rules ensure that this is true. Perhaps the most important is that operations cannot be removed when deriving a new type; they can only be added or replaced. This means that if an operation is primitive for a type then it is necessarily available for all (nonabstract) types in the class rooted at that type. Another vital rule is that we cannot create an object of an abstract type; this prevents dispatching to an abstract subprogram (see 3.6.2). Moreover, as discussed further in 4.5, the tag of an object can never be changed, so the tag of a declared object cannot be changed into the tag corresponding to a type without the appropriate operations.

Another important rule is that type extension is not allowed at a place which is not accessible from the parent type such as within an inner block. This rule ensures that the accessibility of all specific types in a (tagged) class is the same and prevents a value from being assigned to a class wide object and thereby outlive its specific type. A further consequence is that we cannot dispatch to a subprogram which is at an inner level and which might thereby attempt to access non-existent data. Consider

```
package Outer is
   type T is tagged ...;
   procedure P(Object: T);  -- a dispatching operation
   type A is access T'Class;
```

```
      Global: A;
   end Outer;

   procedure Dodgy is
      package Inner is
         type NT is new T with ...;  -- an illegal extension
         procedure P(Object: NT);    -- override
      end;

      package body Inner is
        I: Integer := 0;
        procedure P(Object: NT) is
        begin
           I := I + 1;  -- assign to variable local to Inner
        end P;
      end Inner;
   begin
      Global := new Inner.NT'( ...);
   end Dodgy;

   procedure Disaster is
   begin
      Dodgy;
      P(Global.all);  -- dispatch to non-existent P
   end Disaster;
```

The procedure `Dodgy` attempts to declare the type `NT` and then assign an access to an object of the specific type to the class wide access variable `Global`. If this were allowed then the call of `P` in the procedure `Disaster` would attempt to dispatch to the procedure inside `Inner` and thereby access the variable `I` which no longer exists. Disallowing extensions at an inner level prevents this sort of difficulty.

Note also that having all the types at the same accessibility level ensures that the "subprogram values" in the dispatch table can be implemented just as simple addresses; no level information is required. There is an analogy with access to subprogram values discussed in 3.7.2.

Another important principle is that the dispatch table is the same for all views of a type; in other words there is just one dispatch table common to both a partial view and a full view. However, it can be the case that some operations are not visible from a partial view. This is discussed further in 7.1.1.

Interestingly, it is also possible to have two operations of the same name (and profile), one visible from one view and the other from another view in such a way that they are never both visible from the same view; in this case they would occupy different slots in the dispatch table. These and related possibilities are also illustrated in detail in 7.1.1.

The freezing rules have an important impact on type extension. The basic idea is that a record extension freezes the parent; the key impact is that further primitive operations cannot then be declared for the parent. However, a private extension does not freeze the parent; freezing is postponed until the later full declaration. See 13.5.1.

Finally, we summarize the rules regarding which operations are primitive. The main rule is that only those operations with an operand or result of the type and declared immediately in the package specification with the type declaration (or type extension) are primitive operations. (This general rule applies to both tagged and other types.) Note that if a type is not declared in a package specification then any operations declared in the same declarative region are not primitive and thus not inherited. Because this might give rise to surprises, especially in the case of tagged types, it is in fact forbidden to call a nonprimitive operation in a dispatching way (that is with a class-wide actual); this eliminates the risk of accidentally declaring a tagged type and then finding that what were presumed to be primitive operations do not dispatch.

Note moreover that, in the case of a type extension, although new primitive operations cannot be added except in a package specification, primitive operations inherited from the parent may be overridden wherever the extension is declared and these overridden versions will of course be inherited by any further extension. Thus new slots can only be created by a type declared or extended in a package specification, but existing slots may be overridden wherever a type extension is declared. Consider

```
package P is
   type T is tagged ...;
   procedure Op1(X: T);     -- primitive of T
end P;

with P; use P;
package Q is
   type NTQ is new T with ...;
   procedure Op1(X: NTQ);     -- overrides inherited Op1 from T
   procedure Op2(X: NTQ);     -- additional primitive of NTQ
end Q;

package body P is
   type NTP is new T with ...;
   procedure Op1(X: NTP);     -- overrides inherited Op1 from T
   procedure Op2(X: NTP);     -- not a primitive of NTP
end P;
```

The type `NTQ` is declared immediately inside the specification of package `Q` and thus the operation `Op2` is primitive. On the other hand `NTP` is declared in the body of package `P` and thus although `Op1` overrides the inherited `Op1`, nevertheless the operation `Op2` is not primitive.

These rules are designed to give flexibility with minimum burden. Many type extensions will simply replace existing operations rather than add new ones and it seems a heavy burden to insist that these be in a package specification. Indeed, in the case of the leaves of the tree of types, there is no need to add further primitive operations (if a type is a leaf then any new operation is not inherited by another type and thus there is no need to dispatch); but it is important to be able to override existing operations wherever the type is declared. See the example in 4.4.4.

A minor difference between tagged and nontagged types concerns the parameter modes of overriding operations. In the case of tagged types an overriding operation must have the same parameter modes otherwise dispatching would not work. In the case of nontagged types this does not matter and for compatibility with Ada 83, the modes need not be the same; note that overload resolution ignores parameter modes.

4.4 Examples of Use

This section presents some of the ways in which Ada 95's object oriented programming features may be used and combined with other facilities to address a number of programming paradigms.

An important use of object oriented programming is variant programming. This was amply illustrated by the example of processing alerts in Part One. As we saw, the use of variant records can be both cumbersome and error prone [Wirth 88] whereas the use of type extension is both more flexible and entirely secure.

In this section we give other typical paradigms of use

- An example of different approaches to a standard queue package that can be used as a basis for a wide range of applications.

- An example of a more elaborate heterogeneous doubly linked list abstraction.

- An example showing how alternative implementations can be provided for the same abstraction.
- An example showing how type extension and dispatching can be used to program iterators and similar applications.

4.4.1 Queues

In dealing with the alert example in II.2 we mentioned that the various alerts might be held on a queue ready for processing. Such a queue must have the capability to be heterogeneous because the alerts are of different specific types. This is a common requirement and it is therefore appropriate to develop a package that can be reused for a variety of applications.

However, the strong typing model of Ada 83 made it very difficult to write a common abstraction that could be reused without alteration even through the use of generics. (Only homogeneous structures could be constructed with Ada 83 generic units. Variant records could be used to provide some heterogeneity, but source code changes and possibly extensive recompilation were required to add new variants.)

There are several approaches that can be taken which have a different balance between convenience and efficiency. We will explore a number of these in order to illustrate various considerations and potential pitfalls to be avoided. We will start at the convenient end of the spectrum by considering a package which is generic with respect to the type of data on the queue. The specification might be

```
generic
   type Q_Data(<>) is private;
package Generic_Queues is
   type Queue is limited private;
   function Is_Empty(Q: Queue) return Boolean;
   procedure Add_To_Queue(Q: access Queue; X: in Q_Data);
   function Remove_From_Queue(Q: access Queue) return Q_Data;
   Queue_Empty: exception;
private
   ...
end Generic_Queues;
```

It is important to note that the formal type has an unknown discriminant part. It can then be matched by a specific type or by a class wide type (see 12.5). If we use a specific type then of course the queue is homogeneous but using a class wide type provides a heterogeneous queue. Note also that the exported type `Queue` is limited private; the implementation will inevitably be in terms of pointers to a chained list and making it limited prevents the user from making a copy which might subsequently become nonsense; we will return to the implementation details in a moment.

Values are added to the queue by calling the procedure `Add_To_Queue` and removed by calling the function `Remove_From_Queue`. Making the latter a procedure with profile

```
procedure Remove_From_Queue(Q: access Queue; X: out Q_Data);
```

is rather restrictive because we cannot call the procedure with an uninitialized class-wide object (they are not allowed) and an initialized one will be constrained by its initial value. Such a procedure is therefore only useful if we always know (by some other means) the anticipated specific type of the item being removed. Using a function works because the returned result provides the initial value and thus the constraint.

Incidentally we made the parameter `Q` of the function an access parameter largely because an in out parameter is not allowed for functions. We could have made it an in parameter but the internal implementation of the queue would then need an extra level of indirection. See 6.1.2 for a fuller discussion on the merits of access versus in out parameters. We also have to choose between declaring a queue directly and making it aliased or creating the queue with an allocator. We choose the latter.

So for the alerts, we can write

```
package Alert_Queues is new Generic_Queues(Q_Data => Alert'Class);
use Alert_Queues;
type Queue_Ptr is access all Queue;
The_Queue: Queue_Ptr := new Queue;
...
MA: Medium_Alert := ...;
...
Add_To_Queue(The_Queue, MA);
```

and a value could be retrieved by

```
Any_Alert: Alert'Class := Remove_From_Queue(The_Queue);
```

where the result provides the constraint for `Any_Alert`.

Returning to the example in II.2, we could then call the first form of `Process_Alerts` by

```
Process_Alerts(Any_Alert);
```

and indeed we could directly write

```
Process_Alerts(Remove_From_Queue(The_Queue));
```

It is often preferable to manipulate access values to tagged types rather than tagged type values themselves; partly because this avoids the cost of copying and perhaps more important it overcomes the problem of not knowing the size of the object in the case of a class wide type. So an alternative approach would be to write

```
type Alert_Ptr is access all Alert'Class;
package Alert_Ptr_Queues is new Generic_Queues(Alert_Ptr);
use Alert_Ptr_Queues;
type Queue_Ptr is access all Queue;
The_Queue: Queue_Ptr := new Queue;
...
New_Alert: Alert_Ptr := new Medium_Alert'(...);
...
Add_To_Queue(The_Queue, New_Alert);
```

and then in the body of the second form of `Process_Alerts` we could have

```
Next_Alert: Alert_Ptr := Remove_From_Queue(The_Queue);
...
Handle(Next_Alert.all);
```

This second formulation is very straightforward since the queue is really homogeneous; all the elements are of the same access type and the heterogeneity comes from the nature of the accessed type.

We now return to consider how the generic package might be implemented. An important point is that since the formal type is indefinite, we cannot declare an uninitialized object of the type or a record with a component of the type. This forces us to use dynamic storage. As a first attempt the private part might be

```
private
   type Data_Ptr is access Q_Data;
   type Node;
   type Node_Ptr is access Node;
   type Node is
      record
         D: Data_Ptr;
         Next: Node_Ptr;
      end record;
   type Queue is
      record
         Head: Node_Ptr;
         Tail: Node_Ptr;
      end record;
end Generic_Queues;
```

This is an obvious approach although slightly cumbersome because of the double levels of indirection. This causes a double allocation whenever a new data item is added; one for the node and one for the data itself. Care is also needed in discarding storage when an item is removed. The details are left to the reader.

A problem with the above approach is the encapsulation of the storage management. Although the generic is very reusable it is somewhat costly because of the storage allocation overheads. Of course if the queue were homogeneous and had a definite parameter without <> then it would be simpler because the values could be stored directly; we are paying for the generality. Insisting that the parameter be definite would not be unreasonable because, as shown above, the client can always pass an access type.

A completely different approach is to arrange things so that the user's type provides the storage for the linking mechanism through type extension; this avoids the overheads of storage management but requires a little more effort on the part of the user. Consider the following

```
package Queues is
   type Queue is limited private;
   type Queue_Element is abstract tagged private;
   type Element_Ptr is access all Queue_Element'Class;
   function Is_Empty(Q: Queue) return Boolean;
   procedure Add_To_Queue(Q: access Queue; E: in Element_Ptr);
   function Remove_From_Queue(Q: access Queue) return Element_Ptr;
   Queue_Error: exception;
private
   type Queue_Element is tagged
      record
         Next: Element_Ptr := null;
      end record;
   type Queue is limited
      record
         Head: Element_Ptr := null;
         Tail: Element_Ptr := null;
      end record;
end Queues;
```

The general idea is that the user extends the type `Queue_Element` with the data to be queued. The linking is then done through the private component `Next` of which the user is not aware. The body might be as follows

```
package body Queues is
   function Is_Empty(Q: Queue) return Boolean is
   begin
      return Q.Head = null;
```

```
      end Is_Empty;

      procedure Add_To_Queue(Q: access Queue;
                             E: in Element_Ptr) is
      begin
         if E.Next /= null then
            raise Queue_Error:   -- already on a queue
         end if;
         if Q.Head = null then   -- list was empty
            Q.Head := E;
            Q.Tail := E;
         else
            Q.Tail.Next := E;
            Q.Tail := E;
         end if;
      end Add_To_Queue;

      function Remove_From_Queue(Q: access Queue)
                            return Element_Ptr is
         Result: Element_Ptr;
      begin
         if Is_Empty(Q) then
            raise Queue_Error;
         end if;
         Result := Q.Head;
         Q.Head := Result.Next;
         Result.Next := null;
         return Result;
      end Remove_From_Queue;
   end Queues;
```

Heterogeneous queues can be made because the type `Element_Ptr` is an access to class wide type. There are a number of ways in which this approach can be used which we will now explore using the alert example.

The first point is that we cannot extend the queue element with a class wide type and so we cannot just make a single extension which directly contains any alert. We could of course just extend with a component of the type `Alert_Ptr` and then add and remove alerts as follows

```
   type Alert_Element is new Queue_Element with
      record
         The_Ptr: Alert_Ptr;
      end record;
   ...
   type Queue_Ptr is access all Queue;
   The_Queue: Queue_Ptr := new Queue;

   ...
   New_Alert: Alert_Ptr := new Medium_Alert'(...);
   New_Element: Element_Ptr :=
                  new Alert_Element'(Queue_Element with New_Alert);
   Add_To_Queue(The_Queue, New_Element);
   ...
   Next_Alert := Alert_Element(Remove_From_Queue(The_Queue)).The_Ptr;
```

Note the use of the extension aggregate with the subtype name as the ancestor part, see 3.6.1.

We *could* create distinct element types for each alert level although this has its own problems as will soon become apparent. If we write

```
type Low_Element is new Queue_Element with
   record
      LA: Low_Alert;
   end record;

type Medium_Element is new Queue_Element with
   record
      MA: Medium_Alert;
   end record;
...
```

then adding alerts to the queue is relatively straightforward.

```
MA: Medium_Alert := ...;
New_Element: Element_Ptr :=
                  new Medium_Element'(Queue_Element with MA);
Add_To_Queue(The_Queue, New_Element);
```

Removing an alert in this formulation is less straightforward since we have to identify its specific type by interrogating the tag thus

```
Next_Element: Element_Ptr := Remove_From_Queue(The_Queue);
...
if Next_Element'Tag = Low_Element'Tag then
   Process_Alerts(Low_Element(Next_Element).LA);
```

Unfortunately this brings us back to variant programming which we try to avoid. The essence of the difficulty is that we have dispersed the alerts into the different queue elements and lost their commonality. There are two possible different approaches. The best is to plan ahead and ensure that the complete alert hierarchy is developed with the common queue element already in place. Following II.3, we can write

```
with Queues; use Queues;
package Base_Alert_System is
   type Alert is abstract new Queue_Element with null record;
   procedure Handle(A: in out Alert) is abstract;
end Base_Alert_System;
```

and then we develop all the rest of the alert structure as before. Now all alerts themselves have the linking mechanism already in them and can be directly placed on a queue. So we can now simply write

```
New_Alert: Alert_Ptr := new Medium_Alert'(...);
Add_To_Queue(Queue, New_Alert);
...
Next_Alert := Alert_Ptr(Remove_From_Queue(The_Queue));
```

Note that we have to convert the result to the type `Alert_Ptr`. This conversion requires a runtime check which always passes (because we have only placed alerts on the queue).

An important point to note with this approach is that each element can be on only one queue at a time. An attempt to place an element on a second queue will result in `Queue_Error`. Note that when an element is removed from a queue, its `Next` component is set to **`null`** so that it can then be placed on another queue. Observe that if we consider the elements as like real objects then they can only be in one place at a time and hence only on one queue at a time; so the restriction should not be unrealistic.

If it is quite impossible to modify an existing hierarchy to incorporate the link in the root (perhaps because we do not have the source), then it is still possible to avoid the variant difficulty when removing elements from the queue. The idea is to add a dispatching operation which can extract the particular alert; we can write

```
with Queues; use Queues;
package Alert_Elements is
   type Data_Element is abstract new Queue_Element with null record;
   type Data_Element_Ptr is access all Data_Element'Class;
   function Extract(D: Data_Element) return Alert'Class is abstract;
end Alert_Elements;
```

By introducing the type `Data_Element` we provide a place to attach the required dispatching operation. Note of course that `Extract` only applies to the class rooted at `Data_Element` and not the class rooted at `Queue_Element`.

We can now declare the various types such as `Low_Element` for each alert type as extensions of `Data_Element` and provide an appropriate function `Extract` for each such as

```
type Low_Element is new Data_Element with
   record
      LA: Low_Alert;
   end record;

function Extract(D: access Low_Element) return Alert'Class is
begin
   return D.LA;
end Extract;
```

We can add alerts to the queue much as before but removing alerts is now much simpler. Having copied the pointer to the removed element into `Next_Element` we can then convert to the type `Data_Element` and then call `Extract` thus

```
Next_Element: Element_Ptr := Remove_From_Queue(The_Queue);
Any_Alert: Alert'Class := Extract(Data_Element_Ptr(Next_Element));
```

so that dispatching to the appropriate function `Extract` occurs thereby overcoming the need for variant programming.

Although this mechanism works, it is vulnerable to error if the alert structure is extended. There is a risk that the corresponding extension to the element structure might be forgotten in which case a value of an extended type will not be extracted properly.

We continue this rather long discussion by considering how the original generic queue package could be implemented in terms of the second package. The private part and body might be

```
private
   type Data_Ptr is access Q_Data;
   type Q_Element is new Queues.Queue_Element with
      record
         D: Data_Ptr;
      end record;
   type Queue is new Queues.Queue;
end Generic_Queues;

package body Generic_Queues is
   function Is_Empty(Q: Queue) return Boolean is
   begin
      return Queues.Is_Empty(Queues.Queue(Q));
   end Is_Empty;
```

```
      procedure Add_To_Queue(Q: access Queue; X: in Q_Data) is
      begin
         Queues.Add_To_Queue(Queues.Queue(Q),
            new Q_Element'(Queues.Queue_Element with new Q_Data'(X)));
      end Add_To_Queue;

      function Remove_From_Queue(Q: access Queue) return Q_Data is
      begin
         if Is_Empty(Q) then
            raise Queue_Empty;
         end if;
         declare
            Q_E_P: Queues.Element_Ptr :=
                             Queues.Remove_From_Queue(Queues.Queue(Q));
            D_P: Data_Ptr := Q_Element(Q_E_P.all).D;
            Result: Q.Data := D_P.all;
         begin
            -- can now discard storage occupied by the queue element
            -- and the data; assuming suitable unchecked conversions
            Free(Q_E_P);  Free(D_P);
            return Result;
         end;
      end Remove_From_Queue;
   end Generic_Queues;
```

Note that we have to take care not to lose access to the storage so that it can be freed. In particular the result is copied into a local variable; this is allowed despite the type being indefinite because the variable is initialized. Another point is that `Is_Empty`, `Add_To_Queue` and `Remove_From_Queue` can be slightly simplified since `Queue` is derived from `Queues.Queue` and therefore inherits subprograms with the same identifiers (although different profiles). For example we could simply write

```
      procedure Add_To_Queue(Q: access Queue; X: in Q_Data) is
      begin
         Add_To_Queue(Q,
            new Q_Element'(Queues.Queue_Element with new Q_Data'(X)));
      end Add_To_Queue;
```

The implementation of the generic queue package involves much copying of the data; nevertheless it provides a clean interface and hides all the problems. However, the lower level package is almost as easy to use if the data is structured correctly. Intermediate designs are also possible; for example a generic package that accepts any definite type. The two subprograms could then both be procedures with in out parameters and less indirection would be required.

We conclude with some general observations. It is much easier to manipulate access values when dealing with class wide data. This is largely because of the difficulties of storing such data. We also note that object oriented programming requires thought especially if variant programming is to be avoided. There is a general difficulty in finding out what is coming which is particularly obvious with input-output; it is easy to write dispatching output operations but generally impossible for input.

4.4.2 Heterogeneous Lists

For the next example we consider doubly-linked lists which are a common programming technique.

The implementation shown below uses tagged types and somewhat similar techniques to the second queue package in the last section although at a lower level of abstraction.

```
package Doubly_Linked is

   type Node_Type is tagged limited private;
   type Node_Ptr is access all Node_Type'Class;

   -- define add/remove operations,
   -- assuming head of list is a single Node_Ptr
   procedure Add(Item: Node_Ptr; Head: in out Node_Ptr);
     -- add new node at head of list
   procedure Remove(Item: Node_Ptr; Head: in out Node_Ptr);
     -- remove node from list, update Head if necessary

   -- define functions to iterate forward or backward over list
   function Next(Item: Node_Ptr) return Node_Ptr;
   function Prev(Item: Node_Ptr) return Node_Ptr;
private
   type Node_Type is tagged limited
      record
         Prev: Node_Ptr := null;
         Next: Node_Ptr := null;
         -- other components to be added by extension
      end record;
end Doubly_Linked;
```

This illustrates the specification of a simple doubly linked list abstraction that may be extended with additional components and operations to create useful heterogeneous linked lists. It is similar to the second queue example in that the user extends the type `Node_Type` to contain the required data and it allows heterogeneous lists because the type `Node_Ptr` is an access to class wide type and thus allows the various nodes of different specific types to be linked together.

A difference is that the user refers to the list through the parameter `Head` which is also of the type `Node_Ptr`. Being doubly linked there is no need to separately maintain a reference to the tail of the list. And indeed it is possible to create variations which deal with circular lists.

The procedure `Add` places a new item at the start of the list but in contrast to the queue example, the procedure `Remove` takes the given item from wherever it is in the list. The procedures `Next` and `Prev` enable the user to move over the list as required.

The details of the implementation are not shown but should ensure correct behavior when dealing with an empty list and should also guard against adding an item which is already on the list (or another list) or removing something not on the list.

We can now use the `Doubly_Linked` abstraction to demonstrate programming by extension. We implement a keyed association abstraction using an extension of `Doubly_Linked.Node_Type`. The generic package `Association` takes a `Key_Type`, an equality operation defined on the `Key_Type` and a hash function defined on the `Key_Type`. The exported type `Element_Type` is intended to be further extended with the data to be associated with the key.

```
with Doubly_Linked;
generic
   type Key_Type is limited private;
   with function "="(Left, Right: Key_Type) return Boolean is <>;
   with function Hash(Key: Key_Type) return Integer is <>;
package Association is

   type Element_Type is new Doubly_Linked.Node_Type with
      record
```

```
         Key: Key_Type;
      end record;
   type Element_Ptr is new Doubly_Linked.Node_Ptr;

   function Key(E: Element_Ptr) return Key_Type;

   type Association_Table(Size: Positive) is limited private;
     -- size determines size of hash table

   procedure Enter(Table  : in out Association_Table;
                   Element: in Element_Ptr);

   function Lookup(Table: in Association_Table;
                   Key  : in Key_Type) return Element_Ptr;

   -- other operations on Association_Table (eg, an iterator)...
private
   type Element_Ptr_Array is array (Integer range <>) of Element_Ptr;
   type Association_Table(Size: Positive) is
      record
         Buckets: Element_Ptr_Array(1 .. Size);
      end record;
end Association;
```

An `Association_Table` is a hash table, where each hash value has an associated doubly-linked list of elements. The elements may be of any type derived from `Element_Type`. The head of each list is of the type `Element_Ptr` which is itself derived from `Node_Ptr` (an untagged derived type). All the primitive operations (`Add`, `Remove` etc) which apply to `Node_Ptr` are thus inherited by `Element_Ptr`. The function `Key` returns the key component of the object referred to as parameter.

We can now go on to define a symbol table for a simple language with types, objects, and functions using the association structure. The symbol table allows different types of entries for each of types, objects and functions.

```
with Association;
package Symbol_Table_Pkg is

   type Identifier is access String;
     -- symbol table key is pointer to string
     -- allowing arbitrary length identifiers
   function Equal(Left, Right: Identifier) return Boolean;
   function Hash(Key: Identifier) return Integer;

   -- instantiate Association to produce symbol table
   package Symbol_Association is
     new Association(Identifier, Equal, Hash);
   subtype Symbol_Table is
     Symbol_Association.Association_Table;

   -- define the three kinds of symbol table elements
   -- using type extension
   type Type_Symbol is new Symbol_Association.Element_Type with
      record
         Category: Type_Category;
         Size    : Natural;
      end record;
   type Type_Ptr is access Type_Symbol;
```

```
      type Object_Symbol is new Symbol_Association.Element_Type with
         record
            Object_Type : Type_Ptr;
            Stack_Offset: Integer;
         end record;

      type Function_Symbol is new Symbol_Association.Element_Type with
         record
            Return_Type  : Type_Ptr;
            Formals      : Symbol_Table(5);   -- very small hash table
            Locals       : Symbol_Table(19);  -- bigger hash table
            Function_Body: Statement_List;
         end record;
   end Symbol_Table_Pkg;
```

A type `Symbol_Table` is produced by instantiating the generic `Association` with a key that is a pointer to a string. Then three extensions of `Element_Type` are declared, each of which may be entered into the symbol table. An interesting point is that the elements for the type `Function_Symbol` each themselves contain internal symbol tables.

The body of the generic `Association` package might be as follows

```
   package body Association is
      procedure Enter(Table: in out Association_Table;
                      Element: Element_Ptr) is
                        -- enter new element into association table.
         Hash_Index: constant Integer :=
           (Hash(Element.Key) mod Table.Size) + 1;
         use Doubly_Linked;
      begin
         -- add to linked list of appropriate bucket
         Add(Element, Table.Buckets(Hash_Index));
      end Enter;

      function Key(E: Element_Ptr) return Key_Type is
      begin
         return Element_Type(E.all).Key;
      end Key;

      function Lookup(Table: Association_Table;
                      Key: Key_Type) return Element_Ptr is
                        -- look up element in association table.
         Hash_Index: constant Integer :=
           (Hash(Key) mod Table.Size) + 1;
         Ptr: Element_Ptr := Table.Buckets(Hash_Index); -- head of list
         use Doubly_Linked;
      begin
         -- Scan doubly-linked list for element with
         -- matching key.  Return null if none found.
         while Ptr /= null loop
            if Key(Ptr).Key = Key then
               return Ptr;  -- matching element found and returned
            end if;
            Ptr := Next(Ptr);
         end loop;
         return null;  -- no matching element found
      end Lookup;
```

```
end Association;
```

The operations Enter and Lookup are implemented in a straightforward manner using the operations of the type Element_Ptr inherited from Node_Ptr.

The function Key is interesting. Note first that since Element_Ptr is derived from Node_Ptr its accessed type is also Node_Type'Class (this is a nontagged derivation and when we derive from an access type the accessed type of the derived type is the same as its parent as in Ada 83). So the expression E.**all** is of the type Node_Type'Class. It is then converted to the specific type Element_Type (this is away from the root and so involves a run-time check which will always succeed in this example) and the component is then selected.

Note that since the type Node_Ptr is visible we could declare an object directly and pass an access to it as parameter to the function Key; this would raise Constraint_Error because the function Key is designed to operate on elements and not on nodes in general. We could overcome this by making the types Element_Type and Element_Ptr private so that the underlying relationship to the type Node_Type is hidden.

4.4.3 Multiple Implementations

A very important aspect of object oriented programming is the ability to provide different implementations of the one abstraction. One can do this to some extent in Ada 83 in that one package could have alternate bodies. But only one implementation can be used in one program.

It is worth noting that the possibility of multiple implementations of an abstraction has been recognized for some time [Guttag 77]. However, when abstraction facilities were incorporated into conventional compiled languages, a single implementation per interface was typically adopted for pragmatic reasons [Dijkstra 72]. This is illustrated by CLU [Liskov 77] and Modula [Wirth 77] as well as Ada 83. It was really C++ [Stroustrup 91] that was the first main-stream systems programming language that recognized that the dynamic binding inherent in having objects identify their own implementation could be provided while preserving performance.

Thus, with a true object oriented language, the common structure of the types and their operations provided by inheritance enable different types to be treated as different realizations of a common abstraction. The tag of an object indicates its implementation and allows a dynamic binding between the client and the appropriate implementation.

We can thus develop different implementations of a single abstraction, such as a family of list types [LaLonde 89], matrices (dense or sparse), or set types, as in the next example.

The specification of an Abstract_Sets package might be

```
-- Given
   subtype Set_Element is Natural;

package Abstract_Sets is

   type Set is abstract tagged private;

   -- empty set
   function Empty return Set is abstract;

   -- build set with 1 element
   function Unit(Element: Set_Element) return Set is abstract;

   -- union of two sets
   function Union(Left, Right: Set) return Set is abstract;

   -- intersection of two sets
```

```
   function Intersection(Left, Right: Set) return Set is abstract;

   -- remove an element from a set
   procedure Take(From: in out Set;
                  Element: out Set_Element) is abstract;

   Element_Too_Large: exception;
private
   type Set is abstract tagged null record;
end Abstract_Sets;
```

The package provides an abstract specification of sets. The `Set` type definition is an abstract tagged private type, whose full type declaration is a null record. It defines a set of primitive operations on `Set` that are abstract subprograms. Abstract subprograms do not have bodies and cannot be called directly. However, as primitive operations, they are inherited. Derivatives of `Set` must override these abstract operations to provide their own implementations. Derivatives of `Set` can extend the root type with components providing the desired data representation, and can then implement the primitive operations for that representation.

As an example, one might build an implementation using bit vectors

```
with Abstract_Sets;
package Bit_Vector_Sets is

   type Bit_Set is new Abstract_Sets.Set with private;

   -- Override the abstract operations
   function Empty return Bit_Set;
   function Unit(Element: Set_Element) return Bit_Set;
   function Union(Left, Right: Bit_Set) return Bit_Set;
   function Intersection(Left, Right: Bit_Set) return Bit_Set;
   procedure Take(From: in out Bit_Set;
                  Element: out Set_Element);

private
   Bit_Set_Size: constant := 64;
   type Bit_Vector is
      array (Set_Element range 0 .. Bit_Set_Size-1) of Boolean;
   pragma Pack(Bit_Vector);

   type Bit_Set is new Abstract_Sets.Set with
      record
         Data: Bit_Vector;
      end record;
end Bit_Vector_Sets;

package body Bit_Vector_Sets is

   function Empty return Bit_Set is
   begin
      return (Data => (others => False));
   end;

   function Unit(Element: Set_Element) return Bit_Set is
      S: Bit_Set := Empty;
   begin
      S.Data(Element) := True;
      return S;
```

```
      end;

      function Union(Left, Right: Bit_Set) return Bit_Set is
      begin
         return (Data => Left.Data or Right.Data);
      end;
         ...
   end Bit_Vector_Sets;
```

An alternative implementation more appropriate to very sparse sets might be based on using linked records containing the elements present in a set. We could then write a program which contained both forms of sets; we could convert from one representation to any other by using

```
   procedure Convert(From: in Set'Class; To: out Set'Class) is
      Temp: Set'Class := From;
      Elem: Set_Element;
   begin
      -- build up target set, one element at a time
      To := Empty;
      while Temp /= Empty loop
         Take(Temp, Elem);
         To := Union(To, Unit(Elem));
      end loop;
   end Convert;
```

This procedure dispatches onto the appropriate operations according to the specific type of its parameters. Remember that all variables of class-wide types (such as `Temp`) have to be initialized since class-wide subtypes are indefinite and the tag is given by the tag of the initial value. Note that the equality operators are also dispatching operations so that the expression `Temp /= Empty` uses the equality operation for the type of `From`. Furthermore, assignment is also a dispatching operation although this is not often apparent. In this example, however, if the type of `From` were a linked list then a deep copy would be required otherwise the original value could be damaged when the copy is decomposed. Such a deep copy can be performed by using a controlled type for the inner implementation of the list as explained in 7.4.

Finally, note that the abstract sets package could have been generic

```
   generic
      type Set_Element is private;
   package Abstract_Sets is ...
```

and this would have added an extra dimension for the possibility of reuse.

4.4.4 Iterators

It is a common requirement to wish to apply some operation over all members of a set. One approach was discussed in 3.7.1 using access discriminants. In this section we show a rather different technique using type extension and dispatching. (We start by assuming the example is not generic and consider the impact of genericity later.)

Consider

```
   type Element is ...

   package Sets is
      type Set is limited private;
      ... -- various set operations
```

```
      type Iterator is abstract tagged null record;
      procedure Iterate(S: Set; IC: Iterator'Class);
      procedure Action(E: in out Element;
                       I: in out Iterator) is abstract;
   private
      type Node;
      type Ptr is access Node;
      type Node is
         record
            E: Element;
            Next: Ptr;
         end record;
      type Set is new Ptr;   -- implement as singly-linked list
   end Sets;

   package body Sets is
      ... -- bodies of the various set operations

      procedure Iterate(S: Set; IC: Iterator'Class) is
         This: Ptr := Ptr(S);
      begin
          while This /= null loop
             Action(This.E, IC);  -- dispatch
             This := This.Next;
          end loop;
      end Iterate:

   end Sets;
```

This introduces an abstract type `Iterator` which has a primitive subprogram `Action`. The procedure `Iterate` loops over the set and calls by dispatching the procedure `Action` corresponding to the specific type of the object of the `Iterator` class. The main purpose of the `Iterator` type therefore is to identify by dispatching the particular `Action` to be performed.

The simple example of counting the number of elements in a set can now be written as follows.

```
   package Sets.Stuff is
      function Count(S: Set) return Natural;
   end Sets.Stuff;


   package body Sets.Stuff is

      type Count_Iterator is new Iterator with
         record
            Result: Natural := 0;
         end record;

      procedure Action(E: in out Element;
                       I: in out Count_Iterator) is
      begin
         I.Result := I.Result + 1;
      end Action;

      function Count(S: Set) return Natural is
         I: Count_Iterator;
      begin
         Iterate(S, I);
```

```
      return I.Result;
   end Count;
end Sets.Stuff;
```

The type Count_Iterator is an extension of the abstract type Iterator and the specific procedure Action does the counting. The result is accumulated in a component of the type Count_Iterator and is thereby made accessible to the procedure Action; this component is initialized to zero when the Count_Iterator is declared inside the function Count.

Observe that the type extension is not immediately within a package specification and so it is not possible to add new primitive operations to it. Nevertheless it is possible to override inherited operations such as Action as explained in 4.3. If, for some reason, we wanted to declare additional primitive operations then we would have to introduce an internal package. Note also that we cannot put the type extension inside the function Count because this would break the accessibility rules by making the type extension at a deeper level than the parent type as explained in 3.4.

A further point is that if the parent package Sets were generic with the type Element being a formal parameter as in the example with access discriminants in 3.7.1, then the child package Sets.Stuff would also have to be generic. In that case it would be necessary to move the type extension and the overriding operation Action into the private part of Sets.Stuff for reasons explained in 12.5.

More general actions can be written in a similar manner. Any parameters or results for the action are passed as components in the iterator type. A general procedure to perform some action might be

```
procedure General(S: Set; P: Parameters) is
   I: General_Iterator;
begin
   ...  -- copy parameters into iterator
   Iterate(S, I);
   ...  -- copy any results from iterator back to parameters
end General;
```

and the type General_Iterator and the corresponding Action procedure take the form

```
type General_Iterator is new Iterator with
   record
      ... -- components for parameters and workspace
   end record;

procedure Action(E: in out Element;
                 I: in out General_Iterator) is
begin
   E := ...;  -- do something to element using data from iterator
end Action;
```

It is instructive to compare this example with the corresponding example using access discriminants in 3.7.1. Wherever possible similar identifiers have been used to make the analogy easier. The analogy could be made closer by putting the function Sets.Count of 3.7.1 inside a package as here.

Perhaps the most striking difference is that the two mechanisms are "inside out" to each other in some sense. A notable thing about the access discriminant approach is that the looping mechanism has to be written out for each action. Using type extension the loop is written out once and the dispatching call of Action reaches out to the specific routine required.

The type extension approach has a close similarity to the potential method using an access to subprogram value as a parameter. We would like to write something like

```
procedure Iterate(S: Set;
                  Action: access procedure(E: in out Element)) is
   This: Ptr := Ptr(S);
begin
   while This /= null loop
      Action(This.E);
      This := This.Next;
   end loop;
end Iterate;
```

and then

```
function Count(S: Set) return Natural is
   Result: Natural := 0;

   procedure Count_Action(E: in out Element) is
   begin
      Result := Result + 1;
   end Count_Action;

begin
   Iterate(S, Count_Action'Access);
   return Result;
end Count;
```

but unfortunately we cannot have anonymous access to subprogram parameters as explained in 3.7.2. Declaring a named access type so that the above starts

```
type Action_Type is access procedure(E: in out Element);
...
procedure Iterate(S: Set; Action: Action_Type) is ...
```

does not work either because then the access to the internal procedure `Count_Action` is illegal. We have to make the procedure internal so that it can manipulate the variable `Result`. Note that we would not wish to make `Result` global because that would not work in multitasking programs. See the further discussion in 3.7.2 which also shows how the difficulties can be overcome with generics.

The reason for disallowing more general access to subprogram values is that they would require extra information regarding the environment of the procedure (in this case giving addressability of the variable `Result`). The call of the formal procedure and the dispatching call both serve similar purposes; they enable the iterate procedure to call out to the specific action procedure. In both cases extra information is required; the type extension method enables it to be passed in the type itself. The formal procedure method needs it within the underlying implementation and for a number of reasons this is considered too heavy a burden in the general case.

As mentioned earlier, there is a close analogy between the restrictions which ensure that a procedure value is (nearly) always a single address and those which ensure that a dispatching value is always a single address.

4.5 Dispatching and Redispatching

It is important to understand exactly when dispatching (dynamic binding) is used as opposed to the static resolution of binding familiar from Ada 83. The basic principle is that dispatching is used only when a controlling operand is of a class-wide type. In order to facilitate the discussion we will reconsider the `New_Alert_System` introduced in II.1. The call

```
Handle(A);   -- A of type Alert'Class
```

in the procedure `Process_Alerts` in II.2 is a dispatching call. The value of the tag of `A` is used to determine which procedure `Handle` to call and this is determined at run time.

On the other hand a call such as

```
Handle(Alert(MA));
```

in the procedure `Handle` belonging to the type `Medium_Alert` is not a dispatching call because the type of the operand is the specific type `Alert` as a result of the explicit type conversion.

It is also possible to dispatch on the result of a function when the context of the call determines the tag. Such a result is called a controlling result.

It is an important principle that all controlling operands and results of a call must have the same tag. If they are statically determined then, of course, this is checked at compile time. If they are dynamically determined (for example, variables of a class-wide type) then again the actual values must all have the same tag and of course this check has to be made at run time; `Constraint_Error` is raised if the check fails. In order to avoid confusion a mixed situation whereby some tags are statically determined and some are dynamically determined is not allowed. Thus in the case of the sets example in the previous section, it is illegal to write

```
S: Bit_Set := ...
T: Set'Class := ...
...
S := Union(S, T);   -- illegal
```

even though at run-time it might be the case that the tag of the value of `T` might be `Bit_Set'Tag`. But we could write

```
S := Union(S, Bit_Set(T));
```

and the view conversion will check that `T` is in `Bit_Set'Class` (the tag of `T` does not have to be `Bit_Set'Tag`; it could be of any specific type that can be converted to `Bit_Set`).

A special case arises when the tag is indeterminate. Consider for example the statement

```
To := Empty;
```

in the procedure `Convert`. The parameterless function `Empty` has a controlling result but there is no controlling operand to determine the tag. Consequently the tag is determined from the class-wide parameter `To` which is the destination of the assignment. Of course, the tag of `To` is dynamically determined and this value is used for dispatching on `Empty`. The statement

```
To := Union(To, Unit(Elem));
```

similarly causes dispatching on both `Union` and `Unit` according to the tag of `To`.

Another rule designed to avoid complexity is that it is not legal for a subprogram to have controlling operands or result of different tagged types. Although it is legal to declare two tagged types in the same package, it is not legal to declare a subprogram that has operands or result of both types in that same package. This can, of course, be done outside the package but then the subprogram is not a primitive operation of the types and does not dispatch anyway.

The difficulty with allowing such mixed controlling operands is that it would not be clear how to achieve the various possible combinations of derived operations if both types were derived. If the effect of such mixed operands is required then one type can be replaced by the corresponding class-wide type. See [RM95 3.9.2].

The rules for type conversion (see 3.8) are also designed for clarity. Type conversion is always allowed towards the root of a tree of tagged types and so we can convert a `Medium_Alert` into an `Alert` as in the call

```
Handle(Alert(MA));
```

On the other hand we cannot convert a specific type away from the root (there might be missing components); we have to use an extension aggregate even if there are no extra components. So we can "extend" an `Alert` into a `Low_Alert` by

```
LA := (A with null record);
```

where we have to write **`null record`** because there are no extra components.

We can however convert a value of a class-wide type to a specific type as in

```
MA: Medium_Alert := Medium_Alert(AC);
```

where `AC` is of the type `Alert'Class`. In such a case there is a run-time check that the current value of the class-wide parameter `AC` has a tag that identifies a specific type for which the conversion is possible. Hence it must identify the type `Medium_Alert` or a type derived from it so that the conversion is not away from the root of the tree. In other words we check that the value of `AC` is actually in `Medium_Alert'Class`.

As mentioned in 3.8 some conversions are what is known as view conversions. This means that the underlying object is not changed but we merely get a different view of it.

Almost all conversions of tagged types are view conversions. For example the conversion in

```
Handle(Alert(MA));
```

is a view conversion. The value passed on to the call of `Handle` (that with parameter of type `Alert`) is in fact the same value as held in `MA` but the components relating to the type `Medium_Alert` are no longer visible. And in fact the tag still relates to the underlying value and this might even be the tag for `High_Alert` because it could have been view converted all the way down the tree. Remember also that tagged types are passed by reference.

However, if we did an assignment as in

```
MA := Medium_Alert(HA);
```

then the tag of `MA` would not be changed and would not reflect that of the value in `HA`. All that happens is that the values of the components appropriate to the type of `MA` are copied from the object `HA`. Other components are of course ignored.

Furthermore, if `MA` were not a locally declared variable but an **`out`** or **`in out`** parameter, then again the tag of `MA` would not be changed. Remember, however, that the tag of `MA` in this case need not itself be `Medium_Alert'Tag` since a formal parameter is simply giving a view of the actual parameter and the tag of that could be of any type derived from `Medium_Alert`. But we do know that both sides of the assignment have the components appropriate to `Medium_Alert` and so the assignment works.

Note moreover that conversions of tagged types are allowed as the target of an assignment; thus

```
AC: Alert'Class := ...
...
Medium_Alert(AC) := MA;
```

will check that the tag of `AC` corresponds to `Medium_Alert` or a type derived from it (or in other words checks that `AC` **`in`** `Medium_Alert'Class` is true) and then copies just those components corresponding to the `Medium_Alert` view from the right hand side to the left hand side.

It might help to summarize the golden rules

- the tag of an object never changes; this applies to both specific and class-wide types,

- conversion can never be away from the root, conversion never changes the tag.

The fact that a view conversion does not change the tag is absolutely vital for the implementation of what is known as redispatching.

There are often situations where one would like "multiple dispatch" either within a class, or between two or more classes. Ingalls cites a number of canonical examples such as displaying various kinds of graphical objects on different kinds of displays, event types and handlers, and unification and pattern matching [Ingalls 86]; he suggests a solution for Smalltalk-80 that is more modular than a single dispatch on one parameter, followed by a case statement on the dynamic type of a second parameter. Multiple dispatch is possible in Ada 95 via class-wide types. We first consider the simple case of redispatching within the same class.

It often happens that after one dispatching operation we apply a further common (and inherited) operation and so need to dispatch once more to an operation of the original type. If the original tag were lost then this would not be possible.

Consider again (from II.1)

```
procedure Handle(MA: in out Medium_Alert) is
begin
   Handle(Alert(MA));            -- handle as plain Alert
   MA.Action_Officer := Assign_Volunteer;
   Display(MA, Console);
end Handle;
```

in which there is a call of the procedure `Display`. This call is not a dispatching call because the parameter is of a specific type (and indeed there is only one procedure `Display` which is inherited by all the types).

As written it has been assumed that the display operation is the same for all alerts. However, suppose that in fact it was desired to express the message in different ways according to the level of the alert (in different colors perhaps or flashing).

It would be possible to do this by using the `Tag` attribute to look at the original value of the tag by writing

```
procedure Display(A: Alert; On: Device) is
   AC: Alert'Class renames Alert'Class(A);
begin
   if AC'Tag = Low_Alert'Tag then
      -- display a low alert
   elsif AC'Tag = Medium_Alert'Tag then
      -- display a medium alert
   else
      -- display a high alert
   end if;
end Display;
```

Note that we could have written

```
AC: Alert'Class := A;
```

rather than the renaming but this would cause an unnecessary assignment. Note moreover that we cannot apply the `Tag` attribute to an object of a specific type; it would be rather surprising for `A'Tag` not to be `Alert'Tag`.

However, using tags in this way inside the body of `Display` is quite inappropriate since it has reintroduced the rigid nature of variant programming and could not specifically recognize an alert which is a later extension.

The proper approach is to use redispatching. If we need a different display mechanism for the different alert levels then we write distinct procedures for each one (thus overriding the procedure inherited from the root level) and then redispatch in the various procedures `Handle` as follows

```
procedure Handle(MA: in out Medium_Alert) is
begin
   Handle(Alert(MA));               -- handle as plain Alert
   MA.Action_Officer := Assign_Volunteer;
   Display(Medium_Alert'Class(MA), Console);  -- redispatch
end Handle;
```

This will work properly and the message will be displayed according to the specific type of the original alert.

Another possibility is that the type `Device` might not be represented as a simple enumeration, but instead as a record type, with components representing various aspects of the device. A class of device types could be constructed using tagged types and type extension. Each kind of device must implement an `Output` operation that each kind of alert will use to implement its `Display` operation. In order to call the appropriate `Output` procedure two dispatching operations are involved. First, the type of the alert parameter controls the dispatch to the `Display` procedure, and then within that procedure a dispatch on the `Device` parameter will select the appropriate `Output` operation for the device being used as a display. This double dispatching can be accommodated by making `Display` a class-wide operation of the device class. The `Display` procedure for `Alert` then becomes

```
procedure Display(A: Alert; On: Device'Class) is
begin
   ...
   Output(On);  -- dispatch on On
   ...
end Display;
```

so that within each `Display` procedure, a call to `Output`, with parameter `On` will dispatch to the appropriate operation for the `Device`.

Note once more that it would not have been legal for the specification of `Display` to have been

```
procedure Display(A: Alert; On: Device);
```

since a procedure cannot have controlling operands of more than one tagged type.

4.6 Multiple Inheritance

Some languages permit a derived type, or class, to have more than one parent. These languages are said to support "multiple inheritance". Multiple inheritance is a second-generation object oriented programming mechanism. It originated in MIT's FLAVORS extension to LISP; a precursor to the Common Lisp Object System.

Multiple inheritance poses awkward problems if approached naively, as pointed out by [Budd 91]. There are two conceptual difficulties; what to do if an operation with a given profile belongs to both parents — which, if any, is inherited and how could we distinguish them; and what to do if the same component belongs to both parents from a common ancestor — are there two copies or only one? There are also implementation difficulties associated with these conceptual difficulties.

However, most uses of multiple inheritance fall into one of three idioms each of which can be implemented in Ada 95 using facilities such as access discriminants, generic units and type composition in conjunction with the Ada 95 type extension as will be illustrated in the next few sections.

Given the need to balance the benefits of language defined multiple inheritance with the complexity of the revised language, the potential for distributed overhead caused by multiple inheritance, and the scope of the revision, we chose to support multiple inheritance with a building block approach rather than an extra language construct.

4.6.1 Combining Implementation and Abstraction

The first form of multiple inheritance is, to quote N. Guimaraes of AT&T, "to combine two classes, one that defines the protocol of the component, and another that provides an implementation" [Guimaraes 91]. In languages such as Eiffel and C++, where classes are the only form of module, inheritance is the most common mechanism for combining abstractions. For instance, an Eiffel class `Bounded_Stack[T]`, could be constructed by inheriting from an abstract class `Stack[T]` and a second class `Array[T]`. Class `Array[T]` would then be used to implement the abstract operations not defined by class `Stack[T]`. The programmer must specify the implementation of each such operation, and ideally, the array operations should also be hidden from users of `Bounded_Stack[T]`. The effect of this idiom of multiple inheritance could be achieved in Ada 83 through type composition — inheritance is not required. In Ada, one may implement one type in terms of another, and hide that implementation as a private type.

```
package Bounded is
   type Bounded_Stack(Size: Natural := 0) is private;
   procedure Push(S: in out Bounded_Stack; Element: T);
   procedure Pop(S: in out Bounded_Stack);
   function Top(S: Bounded_Stack) return T;
private
   type T_Array is array (Integer range <>) of T;
   type Bounded_Stack(Size: Natural := 0) is
      record
         Data: T_Array(1..Size);
      end record;
end Bounded;
```

Using the idiom of section 4.4.3 where we discussed the set abstraction, we could derive from a tagged abstract type `Stack`, and implement bounded stacks as arrays. In either case, the operations on `Bounded_Stack` must be explicitly declared, whether being defined or overridden.

4.6.2 Mixin Inheritance

A second idiomatic use of multiple inheritance can be termed mixin inheritance. In mixin inheritance, one of the parent classes cannot have instances of its own and exists only to provide a set of properties for classes inheriting from it. Typically, this abstract, mixin class has been isolated solely for the purpose of combining with other classes. Ada 95 can provide mixin inheritance using tagged type extension (single inheritance) and generic units. The generic template defines the mixin. The type supplied as generic actual parameter determines the parent.

Thus we can write

```
generic
   type S is abstract tagged private;
package P is
   type T is abstract new S with private;
   -- operations on T
private
   type T is abstract new S with
      record
         -- additional components
```

```
      end record;
end P;
```

where the body provides the operations and the specification exports the extended type.

We can then use an instantiation of `P` to add the operations of `T` to any existing tagged type and the resulting type will of course still be in the class of the type passed as actual parameter. Note that in this idiom we have specified both the formal type and the exported type as abstract. This enables the supplied actual type to be abstract. We could declare a cascade of types in this manner thereby adding an unbounded sequence of properties to the original type. We would finally make one further extension in order to declare a type which was not abstract.

As a concrete example, the following generic package adds the property of having multiple versions to any tagged type.

```
with OM;  -- Object Manager provides unique object IDs
with VM;  -- Version Manager provides version control
generic
   type Parent is abstract tagged private;
package Versioned is

   -- A versioned object has an ID, which identifies
   -- the set of versions of that object, plus a version
   -- number that, combined with the ID, identifies an
   -- object uniquely.
   type Versioned_Object is abstract new Parent with private;

   -- given an object, return a new version of that object
   procedure Create_New_Version(O    : in  Versioned_Object
                                New_O: out Versioned_Object);
   -- given an object, returns its version number
   function Version_Number(O: Versioned_Object)
                                         return VM.Version_Number;
   -- given an object and a version number, return that
   -- version of the object
   procedure Get_Version(
     ID_From: in  Versioned_Object;
     Version: in  VM.Version_Number;
     Object : out Versioned_Object);

private

   type Versioned_Object is abstract new Parent with
      record
         ID     : OM.Object_ID := OM.Unique_ID;
         Version: VM.Version_Number := VM.Initial_Version;
      end record;

end Versioned;
```

An important variation on this approach allows us to extend a type privately with generic operations that the client cannot see. This relies on the fact that the full type corresponding to a private extension need not be *directly* derived from the given ancestor. Thus the full type corresponding to

```
type Special_Object is new Ancestor with private;
```

need not be directly derived from `Ancestor`; it could be indirectly derived from `Ancestor`. We can therefore write

```
private
   package Q is new P(Ancestor);
   type Special_Object is new Q.T with null record;
```

and then the type `Special_Object` will also have all the components and properties of the type `T` in the generic package `P`. As written, these are, of course, not visible to the client but subprograms in the visible part of the package in which `Special_Object` is declared could be implemented in terms of them. Note also that the type `Special_Object` is not abstract even though the type `Q.T` is abstract.

As another example of mixin inheritance reconsider the second queue package in 4.4.1. We could make it generic thus

```
generic
   type Data(<>) is abstract tagged private;
package Queues is
   type Queue is limited private;
   type Queue_Element is abstract new Data with private;
   type Element_Ptr is access all Queue_Element'Class;
   function Is_Empty(Q: Queue) return Boolean;
   procedure Add_To_Queue(Q: access Queue; E: in Element_Ptr);
   function Remove_From_Queue(Q: access Queue) return Element_Ptr;
   Queue_Error: exception;
private
```

and then the modified base of the alert system could be

```
with Queues;
package Base_Alert_System is
   type Root_Alert is abstract tagged null record;
   package Alert_Queues is new Queues(Root_Alert);
   subtype Alert_Queue is Alert_Queues.Queue;
   type Alert is
            abstract new Alert_Queues.Queue_Element with null record;
   procedure Handle(A in out Alert) is abstract;
end Base_Alert_System;
```

with the rest of the structure much as before. The major difference is that only alerts can be placed on an alert queue declared as

```
type Alert_Queue_Ptr is access all Alert_Queue;
The_Queue: Alert_Queue_Ptr := new Alert_Queue;
...
```

whereas previously all queues were quite general. With this formulation there is no risk of placing an alert on a queue of some other type such as animals. Thus although the queue is heterogeneous, nevertheless it is constrained to accept only objects of the appropriate class.

This example also illustrates the use of a series of abstract types. We start with `Root_Alert` which is abstract and exists in order to characterize the queues; add the queue element property and thus export `Queue_Element` which is itself abstract; we then derive the abstract `Alert` which forms the true base of the alert system and provides the ability to declare the dispatching operation `Handle`. Only then do we develop specific types for the alerts themselves.

Our final example shows how a window system could be constructed and illustrates the cascade of mixins mentioned above. We start with a basic window and various operations

```
type Basic_Window is tagged limited private;
procedure Display(W: in Basic_Window);
procedure Mouse_Click(W: in out Basic_Window;
```

```
                         Where: in Mouse_Coords);
   ...
```

and then we define a number of mixin generics of the familiar pattern such as

```
generic
   type Some_Window is abstract new Basic_Window with private;
package Label_Mixin is
   type Window_With_Label is abstract new Some_Window with private;
   -- override some operations
   procedure Display(W: in Window_With_Label);

   -- add some new ones
   procedure Set_Label(W: in out Window_With_Label;
                       S: in String);
   function Label(W: Window_With_Label) return String;
private
   type Window_With_Label is abstract new Some_Window with
      record
         Label: String_Quark := Null_Quark;
         -- an X-Windows like unique ID for a string
      end record;
end Label_Mixin;
```

Note that this is slightly different to our previous examples since it can only be applied to the type `Basic_Window` or a type derived from `Basic_Window`.

In the generic body we can implement the overriden and new operations, using any inherited operations as necessary. Thus the new version of `Display` applicable to a `Window_With_Label` might be

```
procedure Display(W: Window_With_Label) is
begin
   Display(Some_Window(W));
   -- display normally using operation of parent type
   if W.Label /= Null_Quark then
      -- now display the label if not null
      Display_On_Screen(XCoord(W), YCoord(W)-5, Value(W.Label));
   end if;
end Display;
```

where the functions `XCoord` and `YCoord` are inherited from `Basic_Window` and give the coordinates for where to display the label.

We might declare a whole series of such packages and then finally write

```
package Frame is
   type My_Window is new Basic_Window with private;
   ...-- exported operations
private
   package Add_Label is new Label_Mixin(Basic_Window);
   package Add_Border is
            new Border_Mixin(Add_Label.Window_With_Label);
   package Add_Menu_Bar is
            new Menu_Bar_Mixin(Add_Border.Window_With_Border);

   type My_Window is
            new Add_Menu_Bar.Window_With_Menu_Bar with null record;
end Frame;
```

Observe that the final declaration has a null extension; it could add further components if required. The various operations exported from the individual mixins can be exported selectively from the package `Frame` by suitable renamings in the package body.

4.6.3 Multiple Views

Finally, there are uses of multiple inheritance where the derived type or class is truly a derivative of more than one parent and clients of that type want to "view it" as any of its parents. This may be accomplished in Ada 95 using access discriminants which effectively enable us to parameterize one record with another.

An access discriminant can be used to enable a component of a record to obtain the identity of the record in which it is embedded (see 3.4.1). This enables complex chained structures to be created and can provide multiple views of a structure. Consider

```
   type Outer is limited private;

private

   type Inner(Ptr: access Outer) is limited ...

   type Outer is limited
      record
         ...
         Component: Inner(Outer'Access);
         ...
      end record;
```

The `Component` of type `Inner` has an access discriminant `Ptr` which refers back to the enclosing instance of the record `Outer`. This is because the attribute `Access` applied to the name of a record type inside its declaration refers to the current instance of the type. This is similar to the way in which the name of a task type refers to the current task inside its own body rather than to the type itself; see [RM83 9.1(4)]. If we now declare an object of the type `Outer`

```
Obj: Outer;
```

then the self-referential structure created is as shown in Figure 4-3. Note that the structure becomes self-referential automatically. This is not the same as the effect that would be obtained with a record in which an instance might happen to have a component referring to itself as a consequence of an assignment. All instances of the type `Outer` will refer to themselves; `Ptr` cannot change because discriminants are constant.

This simple example on its own is of little interest. However, the types `Inner` and `Outer` can both be extensions of other types and these other types might themselves be chained structures. For example, the type `Inner` might be an extension of some type `Node` containing components which access other objects of the type `Node` in order to create a tree. Note in particular that `Inner` could also be

```
type Inner(Ptr: access Outer'Class) is new Node with ...
```

so that heterogeneous chains can be constructed. (`Outer` has to be tagged in this case.) The important point is that we can navigate over the tree which consists of the components of type `Inner` linked together but at any point in the tree we can reach to the enclosing `Outer` record as a whole by the access discriminant `Ptr`.

It should be noted that an access discriminant is only allowed for a limited type. This avoids copying problems with the self-referring components and dangling references.

We now return to the window example of the previous section and show how access discriminants can be used to effectively mix together two hierarchies.

Suppose that as well as the hierarchy of windows which concern areas on the screen, we also have a hierarchy of monitors.

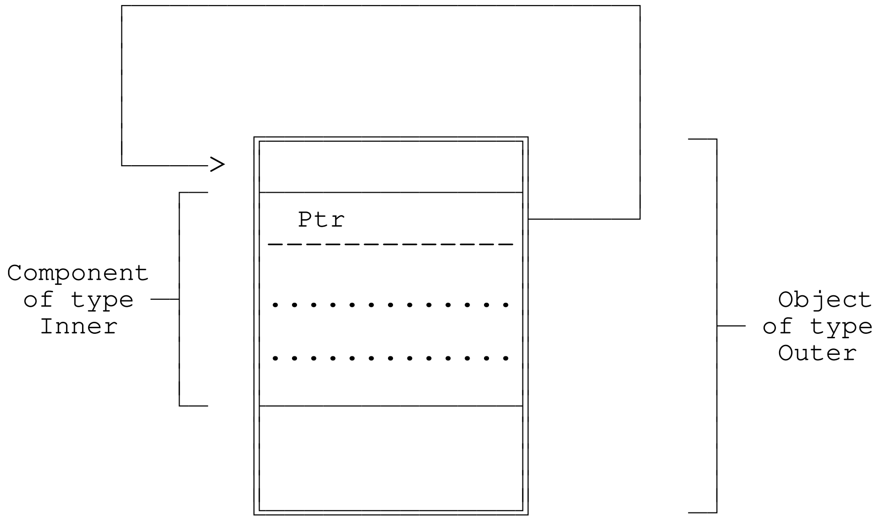

Figure 4-3: A Self-Referential Structure

A monitor is a type which is designed to respond to change; it has a primitive operation `Update` which is called to perform the response. An object that wishes to be monitored keeps a linked list of monitors and calls their `Update` operation whenever necessary; the chain may contain many different monitors according to what might need to be updated. If we were doing a complex modelling application concerned with molecular structure then when we change the object we might wish to redraw some representation on the screen, make a record of the previous state, recompute the molecular weight and so on. The various monitors each contain a reference to the monitored object. The type monitored object itself contains a pointer to the start of the chain and is extended with additional information as needed by the application. Thus we have

```
type Monitor;
type Monitor_Ptr is access all Monitor'Class;

type Monitored_Object is abstract tagged limited
   record
      First: Monitor_Ptr;  -- list of monitors
      -- more components to be added by extension
      -- according to the needs of the specific application
   end record;

type Monitored_Object_Ptr is access all Monitored_Object'Class;

type Monitor is abstract tagged limited
   record
      Next: Monitor_Ptr;
      Obj: Monitored_Object_Ptr;
      -- more components to be added by extension
      -- according to the needs of the specific monitor
   end record;
```

```
procedure Update(M: in out Monitor) is abstract;
...
procedure Notify(MO: Monitored_Object'Class) is
   This_Mon: Monitor_Ptr := MO.First;
begin
   while This_Mon /= null loop
      Update(This_Mon.all);    -- dispatch for each monitor
      This_Mon := This_Mon.Next;
   end loop;
end Notify;
```

where `Notify` is a class wide operation of the type `Monitored_Object` and calls all the `Update` operations of the monitors on the chain. If our object representing the molecule has type `Molecule` then we would write

```
type Monitored_Molecule is new Monitored_Object with
   record
      M: Molecule;
   end record;
...
Proposed_Immortality_Drug: Monitored_Molecule;
```

and then perform all our work on the monitored molecule and from time to time invoke the updates by calling

```
Notify(Proposed_Immortality_Drug);
```

The configuration might be as in Figure 4-4.

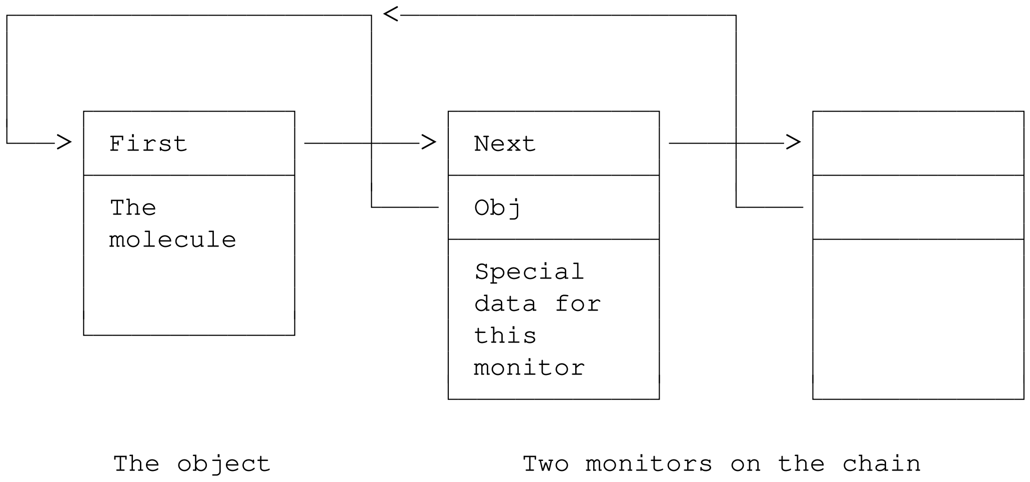

Figure 4-4: A Monitor Chain

Now suppose we want to use one of our windows as part of the updating process so that, for example, the picture of the molecule is displayed within a window rather than directly on the raw screen. In order to do this we need to hang the window display mechanism on the monitor chain so that an appropriate update causes the `Display` operation to be called. In other words we need to create a `Window` that can act as a `Monitor` as well as a `Window`. First we define a mixin that is a monitor and override its `Update` operation thus

```
type Monitor_Mixin(Win: access Basic_Window'Class) is
                            new Monitor with null record;
procedure Update(M: in out Monitor_Mixin);
```

The body for this might be

```
procedure Update(M: in out Monitor_Mixin) is
   -- simply redisplay the window
begin
   Display(M.Win.all);  -- this is a dispatching call
end Update;
```

and now we can mix this `Monitor_Mixin` into any window type by writing

```
type Window_That_Monitors is new My_Window with
   record
      Mon: Monitor_Mixin(Window_That_Monitors'Access);
   end record;
```

where the inner component `Mon` has a discriminant that refers to the outer type. The monitor component of this can now be linked into the chain as shown in Figure 4-5. Calling `Notify` on the monitored molecule results in the various procedures `Update` being called. The `Update` for the type `Monitor_Mixin` calls the `Display` for the type `Window_That_Monitors` of which it is part and this has access to all the information about the window as well as the information about being a monitor.

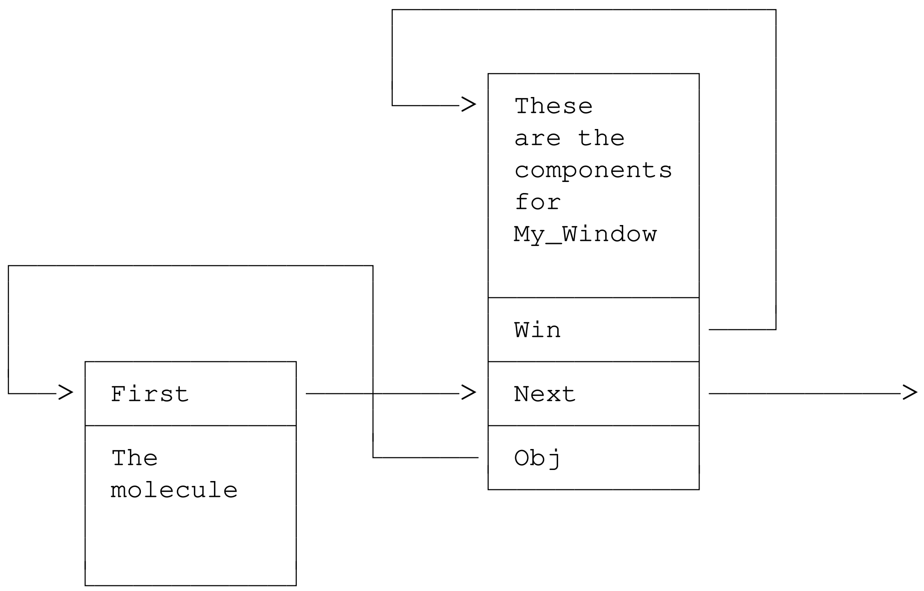

Figure 4-5: The Window-that-Monitors in the Chain

We could of course define a more sophisticated type `Monitor_Mixin` that did other things as well as simply calling the `Display` operation of the associated window.

The examples in this and the previous section show that Ada 95 provides support for the construction of effectively arbitrary multiple inheritance hierarchies. This has been achieved without having intrinsic multiple inheritance which could be a pervasive implementation burden on simple single inheritance applications.

4.7 Relationship with Previous Work

Object oriented programming originated with Simula [Birtwistle 73]. Simula was designed to be an almost upward compatible extension of Algol 60, inspired by the application domain of simulation, although it is really a general purpose programming language. The key insights from simulation were that it is useful to think of a complex simulation as being organized around a collection of autonomous, interacting objects, and that the construction of such simulations could be facilitated by abstracting this notion of object into a language construct.

Simula introduced the notion of a class as an abstraction mechanism over objects. A class is a template for creating objects with a common data structure and operations on that data structure. These operations determine the possible behavior of the objects of the class. Operations may be sensitive to the current state of the object, and may update that state by changing the values of the data structure.

A Simula class definition specifies a data structure for the class, the operations on that data, and a body used to initialize objects of the class upon their creation, like the sequence of statements in a package body. The data definition and procedure declarations constitute the class's interface to programmers. The Simula class is somewhere between a data type and a module. Instances of the class may be declared, assigned to variables, and passed as parameters, like values of a typical data type.

Simula introduced a means to define new classes from old ones; a class could "inherit" from another class, deriving its structure and operations from that "parent". The new class could augment or override its inheritance, adding new data and new operations, or replacing one or more of its operations. Data could not be removed.

Smalltalk [Goldberg 83] was influenced by Simula's notion of class and subclassing. While Simula was a compiled language, Smalltalk was interpreted. It was originally intended as an interactive, systems programming language for Alan Kay's Dynabook project.

Smalltalk introduced the "message-passing" style of invoking operations. A message is a request to an object to invoke an operation. The set of messages that an object recognizes and is capable of responding to is called its "protocol" and is determined by the class of the object. When an object is sent a message, a search begins in the class of the object class for a method (operation definition) corresponding to the message. If not found, the search continues in the parent class (superclass), this continues upward in the class hierarchy until either an appropriate method is found or the root of the hierarchy is reached without success, in which case an error is signaled.

The historical fact that some early object oriented languages were interpreted has contributed to the impression that their mechanisms are necessarily too inefficient for real-time or production use. Many object oriented languages (including Simula) also use implicit reference semantics (in which all variables are really pointers), thereby raising the issue of run-time storage management. It was these efficiency considerations that apparently prevented Ada 83 from providing inheritance and polymorphism, given Ada's overriding concerns with run-time efficiency, and type safety [Brosgol 89]. More recently, there have been a number of languages developed that support object oriented programming in a relatively safe, compiled, and efficient style, including Trellis/Owl [Schaffert 86], Eiffel [Meyer 88], Modula-3 [Nelson 91] and C++ [Ellis 90].

The essence of the evolution of OOPLs has thus been to obtain an appropriate balance between compile-time and run-time identification of the operations to be performed. If the identification is at run-time then the operations are usually called methods; alternative terms are virtual functions (C++, Simula) and dispatching operations (the Ada 95 term).

In Smalltalk-80, for example, method invocations have the form

```
receiver Methodname Argstomethod
```

where `receiver` is the name of the target object.

This syntax simplifies dispatch; the dispatch is determined solely by the class of the receiver of the message. Eiffel and C++ also use the "distinguished receiver" approach.

In languages where a function or procedure call syntax is permitted, and where more than one argument of the call may be of the class, the situation is more complex. Trellis/Owl [Schaffert 86]) follows the Smalltalk-80 tradition and arbitrarily designates the first parameter of the call as determining the dispatch. Some languages distinguish this parameter by its appearance as a prefix in the call.

Other possible schemes include

1 All controlled parameters within the class must share the same type tag.

2 The programmer must select a parameter as the controlling one, as a part of the declaration of the parameter's mode.

3 All controlled parameters must share the same code for the operation (their dispatch tables must all point to the same code body for that operation).

4 The most specific type within the class ("nearest ancestor") applicable to all of the parameters is used.

5 The most general type within the class ("furthest ancestor", the root) applicable to all of the parameters is used.

Ada 95 has adopted (1). This is a logical choice, given that the dispatching operations of a type are the primitive operations of that type and are derived from those of the root type with systematic replacement. So, in Ada 95, more than one operand, or even the result, may control the dispatch. For a primitive operation of a type `T`, the dispatching is controlled by the operands of type `T`, and the result if it is of type `T`.

There are a number of other important differences between Ada 95 and other languages; these differences are designed to add clarity (which encourages programmers to write the correct code) and safety (which prevents disaster if they do not).

The first difference is that in Ada 95, an operation is only dispatching when applied to an actual parameter of a class-wide type. In other OOPLs, a dispatch is possible whenever an object reference or pointer is used as the prefix to the operation. In Ada 95 terms, this means that references/pointers in such OOPLs are always treated as though they designate a class-wide type. Ada 95 allows a formal parameter or an access value to have a specific type as its "referent" (this is the default, preserving upward compatibility and safety). Ada 95 also allows an actual parameter or an access value to have a class-wide type as its referent, in which case dispatching is also possible.

A second difference is that, in Ada 95, if a type `T` is tagged, then all of its primitive operations are dispatching operations; when passed a class-wide operand, they dispatch. In C++, only those particular member functions identified as virtual involve a run-time dispatch. In Ada 95, a (non-dispatching) class-wide operation may be defined by explicitly declaring it with a formal parameter of type `T'Class`. No dispatch is performed in this case, because the body of a class-wide operation expects its actual parameter to still be class-wide. Note that, as in C++, a run-time dispatch may ultimately occur, when such an operation calls a dispatching operation somewhere within its body. This is illustrated by the procedure `Process_Alerts` in II.2.

A final and important difference between Ada 95 and some other OOPLs is that dispatching is safe in the sense that a call to a dispatching operation always has a well-defined implementation to dispatch to. In some OOPLs, such as Smalltalk, it is possible to send a message to an object that has no method for handling that message; a run-time error results. In Ada, such errors are always detected at compile time.

When a primitive operation is called with class-wide operands in all controlling positions, a run-time check is made that all of these controlling operands have the same tag value, and the result is defined to return this same tag value. This common tag value is called the controlling tag value for the call, and identifies the specific type whose corresponding primitive operation is used to implement this call.

This requirement that all controlling operands have the same tag value reflects an existing Ada 83 rule for derived types. The type of all operands of a parent type are systematically replaced with the derived type when inheriting a primitive operation. A primitive operation can only be a primitive operation of one tagged type. It is possible but unusual for a primitive operation to also operate on another type within the same class (but it would not be primitive for that other type). Typically, each primitive operation operates only on one type within the class, and may return this same type.

By treating all controlling operands symmetrically, we avoid some of the difficulties and anomalies encountered in other OOPLs with binary operations. For example, taking the intersection of two sets is viewed as a symmetric operation as opposed to thinking of one set as being special (the "receiver"), with the other set being a mere argument.

By allowing the result context to control the dispatch, we allow parameterless functions to be used to represent type-specific literals, like an empty set in a tagged set class. See the discussion on the procedure `Convert` in 4.5.

There is no need to use run-time dispatch when a controlling operand or result has a statically known specific type. (A mixture of static and dynamically determined tags is not allowed.) In this case, the specific type's implementation of the primitive operation is then called directly (this is effectively a case of "static" binding).

As discussed in 4.3, the canonical implementation model for a type tag is a pointer to a run-time type descriptor, containing pointers to subprogram bodies implementing each of the primitive operations. This implementation model means that the call on a dispatching operation involves only tag-equality checks (if there is more than one controlling operand), and then a call through the appropriate subprogram pointer. The overhead for such a call is bounded, and can be kept to two or three instructions in most cases, ensuring that dispatching operations can be used even in demanding real-time applications. Note that this overhead is typically less than the overhead of using case statements and variant records.

For a tagged type `T`, even the implicitly provided operations (such as `Object'Size` and assignment if nonlimited) use dispatching internally when applied to a class-wide operand, to allow for new components that might be added by type extension.

Generally, for each primitive operation of a parent type, a type extension may either inherit the original implementation, or it may override it. For an operation that had an operand of the parent type, if not overridden it becomes an operation with an operand of the type extension, which simply ignores (and does not affect) the extension part of the operand. However, for an operation that returned a result of the parent type, if not overridden, it becomes an abstract operation that has no implementation for the extension. This is because the extension part of the result would not be defined for such an operation.

Abstract operations allow a type to have a specification for an operation but no implementation for it, effectively requiring that each derivative define its own. Such operations have no default implementation, preventing a derivative from mistakenly inheriting a meaningless implementation. Abstract operations correspond to deferred methods or virtual methods in Smalltalk and C++. The corresponding class is called an abstract superclass.

If a tagged type has an abstract primitive operation, then it must be declared as an abstract type, and no objects with a tag identifying that type may be created. This means that a call to an abstract operation will always dispatch to some non-abstract implementation that is defined for some derivative. No run-time check is needed to detect whether an operation is abstract, because no objects with the tag for an abstract type can ever be created.

To conclude, the model of type extension and polymorphism in Ada 95 combines efficiency of implementation, clarity of program text and security in a cohesive manner. It provides the additional flexibility sought in an object oriented language without compromising the security which was the cornerstone of Ada 83.

4.8 Requirements Summary

The three major study topics

S4.1-A(1) — Subprograms as Objects

S4.3-A(1) — Reducing the Need for Recompilation

S4.3-B(1) — Programming by Specialization/Extension

are directly addressed and satisfied by the facilities discussed in this chapter.

5 Statements

There is naturally very little change in this classical area of the language. The only additional statement is the requeue statement and that is addressed in Chapter 9. There are also additional forms of the delay and select statements and these are also discussed in Chapter 9. The mechanism of assignment including user-defined assignment is closely associated with controlled types and these are discussed in 7.4. The return statement is now moved to Chapter 6 where it properly belongs.

6 Subprograms

Perhaps the most important change to subprograms and their use in Ada 95 is the fact that they are more nearly first class types since they may be manipulated as the target of access types. However, this topic is dealt with in Chapter 3 and we concern ourselves here with other relatively minor improvements to subprograms. These are

- Various aspects of the parameter and result mechanism are improved. The notions of by-copy and by-reference parameters are made more formal. Parameters of mode out may now be read. Subprograms may have parameters of mode out for a limited view of a type.

- A parameter may also be of an anonymous access type.

- A subprogram body may now be provided by renaming; this and other changes increases the categories of conformance rules.

- The rules for new overloadings of `"="` and `"/="` are relaxed.

Other related matters are the calling conventions for interfacing with other languages; these are discussed in Part Three. Abstract subprograms are discussed in Chapter 3.

6.1 Parameter and Result Mechanism

For Ada 95, we define by-copy parameter passing in terms of a subtype conversion and an assignment. This minimizes the number of special rules associated with parameter passing.

For by-reference parameters, the formal parameter is considered a view of the actual.

Certain types are called by-copy types and are always passed by copy. Some other types are called by-reference types and are always passed by reference. For the remaining types the implementation is free to choose either mechanism. Note that the parameter mechanism is independent of the view; thus a private type is always passed by the mechanism appropriate to the full view of the type.

Note that tagged types, task types and protected types are by-reference types.

A similar approach is taken with function results. Certain types are classified as return-by-reference types. Again these include task types and protected types (and most other limited types, see 7.3). In the case of a result returned by reference the function call denotes a constant view of the object denoted by the return expression. In other cases a copy is made. Remember that the result of a function call is treated as an object in Ada 95.

A difference between parameters and results is that tagged types are always by reference as parameters but only returned by reference if limited.

For all modes and both mechanisms of parameters and for results, a subtype conversion is performed if necessary (to provide sliding).

Note in particular that sliding is used for array parameters and results whereas Ada 83 required the more restrictive exact matching of bounds. An array aggregate with **`others`** is still allowed as a parameter or as a result and with the same meaning although for different reasons. In Ada 83 it was allowed because the matching rules provide the bounds whereas in Ada 95 it is allowed because the rules for **others** in assignment are relaxed but there is the overriding rule that aggregates with **others** never slide.

In Ada 95 it is not erroneous to depend on the parameter passing mechanism (by-reference versus by-copy) for those types that allow both, though it is nonportable. This is an example of reducing totally unpredictable behavior (see 1.3).

6.1.1 Out Parameters

In Ada 83 a formal parameter of mode **out** could not be read, even after being initialized within the procedure. This forced certain algorithms to include a local variable just to accumulate a result and which was then assigned to the **out** parameter. Introducing such a local variable is error prone, because the final assignment may be mistakenly omitted.

Similarly, if an **out** parameter is passed to a second procedure to be filled in, the value returned cannot be checked prior to returning from the first procedure.

For Ada 95, we have removed the restrictions on the use of **out** parameters. Specifying that a formal parameter is of mode **out** indicates that the caller need not initialize it prior to the call. However, within the procedure, once the parameter has been initialized, it may be read and updated like any other variable. As with a normal variable, it is an error to depend on the value of an **out** parameter prior to its being initialized.

The added simplicity and flexibility provided by removing the restrictions on reading an **out** parameter allows many of the special cases associated with **out** parameters to be eliminated, including the restriction regarding their use with limited types.

Safety is preserved by ensuring that a subcomponent does not become "deinitialized" by being passed as an **out** parameter. If any subcomponent of a type passed by copy has default initialization, then the whole object is copied in at the start of the call so that the value of such a subcomponent is not lost as a result of a subprogram call during which no assignment is made to the subcomponent. But in practice records are usually passed by reference anyway.

6.1.2 Access Parameters

A formal **in** parameter may be specified with an access definition. Such a parameter is of a general access type that is totally anonymous (has no nameable subtypes), but is convertible to other general access types with the same designated subtype. Access parameters are valuable because they allow dispatching on access values; they are also convenient for use with access discriminants; see 3.7.1.

Access parameters are often an alternative to **in out** parameters especially for tagged types. Thus suppose we have a tagged type and an access type referring to it and appropriate variables such as

```
type T is tagged
   record ...

type Access_T is access T;

Obj: T;

Obj_Ptr: Access_T := new T'(...);
```

plus subprograms taking parameters thus

```
procedure P(X: in out T);

procedure PA(XA: access T);
```

then within the body of both P and PA we have read and write access to the components of the record. Indeed because of automatic dereferencing the components are referred to in the same way. And since tagged types are all always passed by reference and never by copy, the effect is much the same for many situations. For example dispatching is possible in both cases. However, there are a number of important differences.

In the case of the **in out** parameter, the actual parameter could be Obj or Obj_Ptr.**all** thus

```
P(Obj);  P(Obj_Ptr.all);
```

whereas in the case of the **access** parameter, the actual parameter has to be Obj'Access or Obj_Ptr. Moreover in the former case the variable Obj must be marked as **aliased**

```
Obj: aliased T;
PA(Obj'Access);  PA(Obj_Ptr);
```

Remember also that an actual parameter corresponding to an access parameter cannot be null. Moreover, accessibility checks for access parameters are dynamic and the parameter carries with it an indication of its accessibility.

A vital difference is that a function cannot have an in out parameter and so an access parameter is essential if we need a function. We recall from 4.4.1 that the alternative of a procedure with an out parameter corresponding to the function result is not possible in some cases such as where the result type is class wide and we do not know the anticipated specific type.

Other important considerations occur if we have a sequence of nested calls. Thus suppose we have other procedures Q and QA and that in their bodies we wish to call the procedures P and PA with the parameter passed on. There are four possible combinations of calls to consider. We have

```
procedure Q(X: in out T) is
begin
   P(X);                  -- in out passed on to in out
   ...
   PA(X'Access);          -- in out passed on to access
end Q;

procedure QA(XA: access T) is
begin
   P(XA.all);             -- access passed on to in out
   ...
   PA(XA);                -- access passed on to access
end QA;
```

All of these calls are legal. The call of PA from within Q is legal because formal parameters of a tagged type are considered aliased and treated just like an aliased local variable. Hence X'Access is allowed but the accessibility level is that of a variable local to Q. This means that the accessibility level passed to PA indicates that Q.X is local to Q and will not reflect the accessibility level of the original actual parameter passed to Q (and of course that information was not passed to Q anyway).

In the reverse situation where QA calls P no accessibility information is passed on. Moreover, it should be noted that the parameter XA.**all** creates a view of the original parameter and this view is passed on; no local object is created. Thus the information passed on is simply the original "reference" minus the accessibility information.

The uniform cases where Q calls P or QA calls PA are straightforward. The parameter is passed on unchanged, in the first case there is no accessibility information and in the second it is passed on intact.

There is a lot of merit in using access parameters because they pass the correct accessibility level and avoid the risk of an illegal program or unexpectedly raising Program_Error when attempting a conversion to a named access type. For example, assuming T and Access_T are declared at the same level as the procedures and that Q and QA are called with actual parameter Obj or Obj'Access respectively, then it would be illegal to write

```
Obj_Ptr := Access_T(X'Access);
```

inside Q since the static accessibility check fails, whereas it is legal to write the corresponding

```
Obj_Ptr := Access_T(XA);
```

inside QA, and the dynamic accessibility check succeeds.

Furthermore if we also had a procedure

```
procedure RA(XA: access T) is
begin
   ...
   Obj_Ptr := Access_T(XA);
   ...
end RA;
```

in which a similar conversion is performed, then calling RA from Q by

```
RA(X'Access);
```

results in raising Program_Error on the attempted conversion in RA whereas calling RA from QA by

```
RA(XA);
```

works successfully because the original accessibility level is preserved.

However, remember that we can always use Unchecked_Access which avoids the accessibility checks. This would overcome the difficulties in the above examples but of course the use of Unchecked_Access in general can result in dangling references; the responsibility lies with the user to ensure that this does not happen.

For a further discussion on access parameters and a comparison between them and parameters of a named access type see 3.7.1.

6.2 Renaming of Bodies and Conformance

In Ada 95, we allow a subprogram body to be provided by renaming another subprogram. This is a great convenience in those many cases in Ada 83 where the programmer was forced to provide a body which simply called some other existing subprogram. In order that the implementation can be just a jump instruction, the subprogram specification must be "subtype conformant" with the body used to implement it. Subtype conformance is required because the caller sees only the subprogram specification, and therefore has prepared the parameters, performed constraint checks, and followed the parameter passing conventions determined by the specification.

Several different rules existed in Ada 83 governing matching between subprogram specifications. For the purposes of hiding the name of the subprogram, only the types of the formal parameters and the result, if any, were relevant (see [RM83 8.3(15)]). For renaming and generic instantiation, the modes also had to match (see [RM83 8.5(7)] and [RM83 12.3.6(1)]). Between a specification and its body, syntactic equivalence was required ([RM83 6.3.1(5)]).

For Ada 95, in order to support access to subprogram types, and to support the provision of a body by a renaming, an intermediate level of matching is needed. This intermediate level requires static subtype matching, but allows formal parameter names and defaults to differ.

To improve the presentation in Ada 95, the descriptions of these various levels of subprogram matching are gathered into the section on Conformance Rules [RM95 6.3.1]. Each level of matching is given a name and they can be arranged in a strictly ascending order of strength as follows

Type conformance. This is the matching that controls hiding.

Mode conformance. This is the matching required for renaming and generic formal subprograms.

Subtype conformance. This is the matching required for access to subprogram types, and for specifying a body via renaming.

Full conformance. This is the matching required between the declaration and body of a subprogram, and between multiple specifications of a discriminant part.

In addition to centralizing these definitions, we have also relaxed the full conformance rules, in order to make them represent static semantic equivalence, rather than syntactic equivalence. This has the effect of eliminating certain anomalies (such as the non-transitivity of Ada 83 conformance), as well as being more natural for the programmer and easier to implement.

6.3 Overloading of Equality and Inequality Operators

In Ada 83, the "=" operator could be explicitly defined only for limited types (other than via a devious method based on a curious loophole in generic instantiation). This restriction was justified largely on methodological grounds. However, experience with Ada has illustrated several circumstances where it is very natural to provide a user-defined equality operator for nonlimited types. For example, within a three-value logic abstraction, `"="` should return either `True`, `False`, or `Unknown`. For vector processing, it is natural to define a component-wise `"="` operator for vectors, producing a vector of Boolean values as the result. In such cases, it is also important to be able to explicitly define `"/="`, since it is not the simple Boolean complement of `"="`.

In Ada 95, we allow `"="` to be treated like any other relational operator. But note that when the result type of a user-defined `"="` operator is `Standard.Boolean`, a complementary definition for `"/="` is automatically provided. Explicit definitions for `"/="` are also permitted, so long as the result type is not `Standard.Boolean`. A `"/="` operator with a result type of `Standard.Boolean` may thus become defined only as an implicit side-effect of a definition for `"="`.

Of course, dispatching can occur on `"="`. For example in the procedure `Convert` in 4.4.3, the comparison in

```
while Temp /= Empty loop
```

dispatches. Typically the equality will be predefined but of course it might not be.

6.4 Requirements Summary

Many of the changes in this chapter are consequences of the general requirement

R2.2-C(1) — Minimize Special Case Restrictions discussed in [DoD 90 A.3]; this lists three examples which have been met by this chapter. They are the ability to redefine `"="`, the ability to read **`out`** parameters and the use of sliding (array subtype conversion).