Genericity versus Inheritance

Bertrand Meyer

University of California, Santa Barbara
and Interactive Software Engineering, Inc.
Current address: Interactive Software Engineering, Inc.
270 Storke Road, Suite #7, Goleta CA. 93117

ABSTRACT

Genericity, as in Ada or ML, and inheritance, as in object-oriented languages, are two alternative techniques for ensuring better extendibility, reusability and compatibility of software components. This article is a comparative analysis of these two methods. It studies their similarities and differences and assesses to what extent each may be simulated in a language offering only the other. It shows what features are needed to successfully combine the two approaches in a statically typed language and presents the main features of the programming language Eiffel, whose design, resulting in part from this study, includes multiple inheritance and a limited form of genericity under full static typing.

1 - OVERVIEW

In spite of its name, today's software is usually not *soft* enough: adapting it to new uses turns out in most cases to be a harder endeavor than should be. It is thus essential to find ways of enhancing such software quality factors as extendibility (the ease with which a software system may be changed to account for modifications of its requirements), reusability (the ability of a system to be reused, in whole or in parts, for the construction of new systems) and compatibility (the ease of combining a system with others).

Good answers to these issues are not purely technical, but must include economical and managerial components as well; and their technical aspects extend beyond programming language features, to such obviously relevant concerns as specification and design techniques. It would be wrong, however, to underestimate the technical aspects and, among these, the role played by proper programming language features: any acceptable solution must in the end be expressible in terms of programs, and programming languages fundamentally shape the software designers' way of thinking.

This article is a comparative analysis of two classes of programming language features for enhancing extendibility, reusability and compatibility. It assesses their respective strengths and weaknesses, examines which of their components are equivalent and which are truly different, shows how the two approaches complement each other, and explains how they have been combined in a particular programming language design.

The two approaches studied are *genericity* and *inheritance*; both address the above issues by allowing the definition of flexible software elements amenable to extension, reuse and combination. The first is a technique for defining elements that have more than one interpretation, depending on parameters representing types; the second makes it possible to define elements as extensions or restrictions of previously defined ones.

Both methods apply some form of *polymorphism*, a notion that may be defined as the ability to define program entities that may take more than one form. A simple form of polymorphism, used in both cases, is *overloading*, the ability to attach more than one meaning to the same name, ambiguities being resolved by examining the context of each occurrence of the name, either at compile time (for statically typed languages) or at run time.

Although the two approaches may be applied outside the strict realm of programming, for example to specification or design languages, we shall confine our study to programming languages. In this field, genericity is most notably present in Ada; inheritance is a feature of object-oriented languages and was introduced by Simula 67.

Ada and object-oriented languages have until now aroused interest in rather different communities and it is not surprising that no comparative analysis seems to have been published. (The only related work that we know of is the as yet unpublished, more theory-oriented article by Cardelli and Wegner [6], of which we became aware as this paper was going to press). However we feel that beyond "cultural" differences the real goals pursued are the same, so that it is fruitful to perform an in-depth comparison of the technical solutions obtained on both sides.

Section 2 introduces genericity; section 3 discusses inheritance; sections 4 and 5 compare the two approaches by studying whether the effect of each may be achieved with the other; section 6 describes how a particular programming language, Eiffel, uses a balanced combination of the two techniques. Section 7 summarizes the results achieved.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1986 ACM 0-89791-204-7/86/0900-0391 75¢

2 - GENERICITY

Genericity as offered by Ada is present in few other programming languages (examples include CLU[1] [10] and LPG [2]), but is offered by several formal specification languages, such as Z [1], Clear [5] OBJ2 [9] and LM [12]. A variant of this approach was developed in connection with the language ML [11, 7] and has been integrated into a number of functional languages.

We shall concentrate on the Ada form, restricting ourselves to *type genericity*, that is to say the ability to parameterize a software element (in Ada, a package or subprogram) by one or more types. Generic parameters have other, less interesting uses in Ada, such as parameterized dimensions for arrays.

We shall distinguish between *unconstrained* genericity, whereby no specific requirement is imposed on generic parameters, and *constrained* genericity, whereby a certain structure is required.

2.1 - Unconstrained genericity

In its simplest form, unconstrained genericity may be seen as a technique to bypass the unnecessary requirements imposed by static type checking.

Consider the example of a simple procedure for exchanging the values of two variables. In a language which is not statically typed, such as Lisp, we would write something like the following, syntactic differences notwithstanding:

/1/
```
procedure swap (x, y) is
   t: local
begin
   t := x; x := y ; y := t;
end swap
```

The type of the elements to be swapped and of the local variable *t* does not have to be specified. However this may be too much freedom since a call of the form *swap* (*a*, *b*), where *a* is, say, an integer, and *b* a character string, will not be prohibited even though it is probably an error.

Statically typed languages such as Pascal address this problem by requiring programmers to declare explicitly the types of all variables and formal parameters; they enforce a statically checkable type compatibility requirement between actual and formal parameters in calls, and between source and target in assignments. In such a language, the procedure to exchange the values of two variables of type *T* becomes:

/2/
```
procedure T_swap (x, y: in out T) is
   t: T
begin
   t := x; x := y ; y := t;
end swap
```

Demanding that *T* be specified as a single type averts type incompatibility errors, but has the unpleasant consequence of requiring a new procedure declaration for each type for which a swap operation is needed; in the absence of overloading, a different name must be assigned to each such procedure, for example *int_swap*, *str_swap* and so on. Such multiple declarations lengthen and obscure programs. The example chosen is particularly bad since all the declarations will be identical except for the two occurrences of *T*.

Static typing may be considered too restrictive here: the only real requirement is that the two actual parameters passed to any call of *swap* should be of the same type, and that their type should also be applied to the declaration of the local variable *t*.

A language with genericity provides a tradeoff between too much freedom, as with untyped languages, and too much restraint, as with Pascal. In such a language, one may declare *T* as a generic type parameter to the *swap* procedure. In quasi-Ada, the procedure may be declared as follows:

/3/
```
generic
   type T is private;
procedure swap (x, y: in out T) is
   t: T
begin
   t := x; x := y ; y := t;
end swap
```

The only difference with real Ada is that we have merged together, for ease of presentation, the two parts of an Ada subprogram declaration, header and body; their separation in Ada comes from a concern for information hiding, orthogonal to this discussion.

The **generic**... clause introduces type parameters. By specifying *T* as a "private" type, the writer of this procedure allows himself to apply to objects of type *T* (*x*, *y* and *t*) operations available on all types, such as assignment or comparison, and these only.

A declaration such as the above does not actually introduce a procedure but rather a procedure pattern; actual procedures will be obtained by instantiating the pattern with actual type parameters, as in:

/4/
```
procedure int_swap is new swap (INTEGER);
procedure str_swap is new swap (STRING);
```

etc. Now assuming that *i* and *j* are variables of type *INTEGER*, *s* and *t* of type *STRING*, then of the following calls

```
int_swap (i, j);
str_swap (s, t);
int_swap (i, s);
str_swap (s, j);
str_swap (i, j);
```

only the first two will be legal; the other are statically incorrect.

More interesting than parameterized subprograms are parameterized packages. Ada packages (and their

equivalents in other modular languages, such as modules in Modula 2) are syntactical encapsulations of groups of related program entities such as subprograms, types and variables. One of the most important applications of packages, and the only one considered in this article, is data abstraction: each package contains the implementation of a type and of the operations applicable to elements of that type.

Ada packages may be declared with generic parameters. For example, the following generic package describes stacks of elements of an arbitrary type *T*:

/5/
```
generic
   type T is private;
package STACKS is
   type STACK (size: POSITIVE) is
      record
         space: array (1..size) of T;
         index: NATURAL
      end record;
   function empty (s: in STACK) return BOOLEAN;
   procedure push (t: in T; s: in out STACK);
   procedure pop (s: in out STACK);
   function top (s: in STACK) return T;
end STACKS
```

We have given only the public part ("specification") of the package; the package implementation ("body"), which describes the subprogram bodies, must be declared separately. For technical reasons having to do with the problems of Ada compilation, the implementation of the types supported by a package, such as *STACK* here, is given in the public part. For information hiding purposes, this implementation may be given in the **private** clause of the public part, a kind of purgatory between specification and body; however we do not need to use this feature for the present discussion.

As with generic subprograms, the above does not define a package but a package pattern; actual packages may be obtained by instantiation, as in

/6/
```
package INT_STACKS is new STACKS (INTEGER);
package STR_STACKS is new STACKS (STRING);
```

In a program unit that has access to both of these instances of *STACKS*, dot notation may be used to distinguish between namesake elements: for example the type "stack of integers" will be denoted by *INT_STACKS.STACK*, and the type "stack of strings" by *STR_STACKS.STACK*; the corresponding "push" procedures are *INT_STACKS.push* and *STR_STACKS.push*.

We may note again the compromise that generic declarations achieve between typed and untyped languages. *STACKS* provides a pattern for the declaration of modules implementing stacks of elements of all possible types *T*, while retaining the possibility to enforce type checks: for example it will not be possible to push an integer onto a stack of strings.

Both examples above (swap and stack) evidence a form of genericity which we call ***unconstrained*** since there is no specific requirement on the types that may be used as actual generic parameters. In the first case, one may swap the values of variables of any type; in the second, one may create stacks of values of any type, provided all values in a given stack are of the same type.

In other cases, however, a generic definition will only be meaningful if the actual generic parameters satisfy some conditions. We define this form of genericity as ***constrained***.

2.2 - Constrained genericity

As with unconstrained genericity, we consider two constrained examples: first a subprogram and then a package.

Assume we want to define a generic subprogram for finding the minimum of two values. Using the pattern of ***swap*** above, we may write the following function:

/7/
```
generic
   type T is private;
function minimum (x, y: T) return T is begin
   if x <= y then return x;
   else return y end if;
end swap
```

However such a function declaration is not always meaningful: it should only be instantiated for types *T* on which a comparison operator <= is defined. In an untyped language we might defer checking of this property until run-time, but this is not acceptable in a language that enhances security through static typing. We need a way to specify that type *T* must be equipped with the right operation.

In Ada this will be written by treating the operator <= as a generic parameter of its own. Syntactically it will be a function; note that, as a syntactic facility, Ada makes it possible to declare functions to be invoked in infix form (as <=) by declaring them with a name enclosed in double quotes, for example "<=" in the case at hand. Again the following declaration becomes legal Ada if the public part and implementation are taken apart.

/8/
```
generic
   type T is private;
   with function "<=" (a, b: T)
      return BOOLEAN is <>;
function minimum (x, y: T) return T is
begin
   if x <= y then return x;
   else return y end if;
end swap
```

The keyword **with** is used to introduce generic formal parameters representing subprograms, such as the function "<=".

This declaration may now be instantiated as follows for a type, say *T1*, for which a function, say *T1_le*, of type **function** (*a*, *b*: *T1*) **return** *BOOLEAN* is defined:

/9/
```
function T1_minimum is new minimum (T1, T1_le);
```

If, on the other hand, the *T1_le* function is in fact called "<=", that is to say if its name and type match those of the corresponding formal subprogram, then one may omit it from the list of actual parameters to the generic instantiation of the subprogram. For example, the type *INTEGER* has a predefined "<=" function with the right type, so that we can simply declare

/10/
```
function int_minimum is new minimum (INTEGER);
```

This ability to use default actual subprograms with matching names and types is obtained by specifying **is <>** in the declaration of the formal generic subprogram, as was done above with "<=". Note that the overloading of operators, as permitted (and in fact encouraged) by the design of Ada, plays an essential role here: "<=" may be defined for many different types.

This discussion of constrained genericity in the subprogram case readily applies to generic packages. Assume that we want to write a generic matrix manipulation package, applicable to matrices of objects of any type *T*, with matrix sum and product as basic operations. Such a definition is only meaningful if type *T* has a sum and a product of its own, and each of these operations has a zero element; these features of *T* will be needed in the implementation of matrix sum and product. The public part of the matrix package may be written as follows:

/11/
```
generic
   type T is private;
   zero: T;
   unity: T;
   with function "+" (a, b: T) return T is <>;
   with function "*" (a, b: T) return T is <>;
package MATRICES is
   type MATRIX (lines, columns: POSITIVE) is
      array (1..lines, 1..columns) of T;
   function "+" (m1, m2: MATRIX) return MATRIX;
   function "*" (m1, m2: MATRIX) return MATRIX;
end MATRICES;
```

Instances of this package may be obtained as follows:

/12/
```
package INT_MATRICES is
   MATRICES (INTEGER, 0, 1);
package BOOL_MATRICES is
   MATRICES (BOOLEAN, false, true, "or", "and");
```

As in the previous example, actual subprogram parameters (corresponding here to "+" and "*") may be omitted for a type such as *INTEGER* which possesses matching operations; however they must be specified for *BOOLEAN*. (It is convenient to declare optional parameters as the last ones in the formal parameter list; otherwise a keyword notation must be used when the corresponding actual parameters are omitted).

It is interesting here to show how the implementation part of such a package will look. It is enough to give one of the function bodies in this package; we take matrix product as an example.

/13/
```
package body MATRICES is
   .......... other declarations ............
   function "*" (m1, m2: T) is
      result: MATRIX (m1'lines, m2'columns);
   begin
      if m1'columns /= m2'lines then
         raise INCOMPATIBLE_SIZES;
      end if;
      for i in m1'RANGE(1) loop
         for j in m2'RANGE(2) loop
            result (i, j) := zero;
            for k in m1'RANGE(2) loop
               result (i, j) :=
                  result (i, j) + m1 (i, k) * m2 (k, j)
            end loop;
         end loop;
      end loop;
      return result
   end "*";
end MATRICES;
```

Three comments are in order for the reader not familiar with all the details of Ada:

- for a parameterized type such as *MATRIX* (*lines, columns: POSITIVE*), a variable declaration must provide actual values for the parameters, e.g. *mm: MATRIX* (*100, 75*); these values may then be retrieved using the apostrophe notation as in *mm'lines* which in this case has value 100;
- if *a* is an array, *a'RANGE*(*i*) denotes the range of values in its *i*-th dimension; for example *m1'RANGE*(*1*) above is the same as *1..m1'lines*;
- if requested to multiply two dimension-wise incompatible matrices, the program raises an exception; it does not execute the code that follows the **raise** instruction. The package should include code to handle the exception.

The two examples given (minimum and matrices) are representative of the Ada techniques for constrained genericity. They also show a serious limitation of languages such as Ada in this area: the fact that only syntactic constraints may be expressed. All that a programmer may require is the presence of certain subprograms ("<=", "+", "*" in the examples) with given types; but the declarations are meaningless unless some semantic constraints are also satisfied. For example, *minimum* only makes sense if "<=" is a partial order relation on *T* (reflexive, antisymmetric, transitive); and the *MATRICES* package should not be instantiated for a type *T* unless the operations "+" and "*" not only have the right type ($T \times T \longrightarrow T$) but also give *T* the structure of a ring (associativity, distributivity, *zero* a zero element for "+" and *unity* for "*", etc).

To include such formal constraints, one has to leave the realm of programming languages such as Ada for such specification languages as Clear and OBJ2 (the latter executable) or the experimental programming language LPG.

2.3 - Implicit genericity

It is important to mention a form of genericity quite different from the above Ada-style explicit parameterization: the implicit polymorphism exemplified by the work on the ML functional language [14, 7].

This technique is based on the remark that explicit genericity, as seen above, places an unnecessary burden on the programmer, who must give generic types even when the context provides enough information to deduce a correct typing. It may be argued, for example, that the very first version (/1/) given for procedure **swap**, with no type declaration, is acceptable as it stands: with adequate typing rules, a compiler has enough information to deduce that x, y and t must have the same type. Why not then let programmers omit type declarations when they are not strictly needed conceptually, and have the compiler check that all uses of an identifier are consistent?

This approach, sometimes called "unobtrusive type checking" [15], attempts to reconcile the freedom of untyped languages with the security of typed ones. It has been elegantly implemented in ML and other functional languages. One may argue, of course, that some obtrusiveness may be useful: the redundancy entailed by explicit type declarations may enhance program readability. Whatever the answer to this debate may be, the question of explicit or implicit genericity is not directly connected to the present discussion; for the purposes of comparison with inheritance, both forms of genericity are somehow equivalent.

Without committing ourselves as to which form is best, we have chosen to rely on the explicit form exemplified by Ada, which, for our study, has the obvious advantage that generic parameters stand out more visibly.

3 - INHERITANCE

The inheritance technique was introduced in 1967 by Simula 67 [3, 8, 11]. It has been widely imitated in other object-oriented languages.

As with genericity, we will mostly introduce this technique through examples. Since we need a notation, we shall rely on a particular one, that of the object-oriented language Eiffel [13].[1] Much of the discussion would readily transpose to other object-oriented languages; however Eiffel's emphasis on static typing, and its design as an object-oriented language for actual software engineering applications (as opposed to, say, artificial intelligence or exploratory programming) make it particularly suitable for this discussion. Only the elements of Eiffel which are essential to this article are introduced; more details may be found in the reference quoted.

The fundamental idea of inheritance is that new software elements may be defined as extensions of previously defined ones; existing elements do not have to be modified when used as a basis for new definitions.

[1] Eiffel and the associated compilers and tools are products of Interactive Software Engineering, Inc., Goleta (California).

This concept blends particularly well with the object-oriented approach, in which basic software elements are implementations of abstract data types: the extensions of software elements mentioned above will then correspond to refinements of hierarchies of abstract data types.

The basic tenet of object-oriented programming languages may be described as the idea that the fundamental elements, modules, are not only associated with implementations of abstract data types (an effect which may be achieved in any language offering modular features and information hiding, such as Ada or Modula 2), but *are* such implementations. In other words, the defining equality of object-oriented languages is

Module* ≡ *Type

This dogmatic identification of two apparently distinct programming notions, one syntactic, the other semantic, may appear too strict and indeed has some disadvantages. But it also gives object-oriented programming languages and the associated design method a strong conceptual integrity, and provides powerful techniques for satisfying the software quality requirements mentioned above.

As an example of such a module-type, called a class in Eiffel as in Simula and many other object-oriented languages, consider the following outline of an implementation of "special files" in the Unix sense, that is to say, files associated with devices:

/14/
```
class DEVICE export
    open, close, opened
feature
    open (file_descriptor: INTEGER) is
        do
            .........
        end; -- open
    close is
        do
            .........
        end; -- close
    opened: BOOLEAN
end -- class DEVICE
```

This class is the implementation of an abstract data type characterized by three "features", ***open***, ***close*** and ***opened***. There are two kind of features: attributes and routines. Routines, like ***open*** and ***close*** here, are operations applicable to objects of the class; routines are further divided into procedures which, as the two shown here, perform some actions, and functions (seen in later examples), which return a value. Attribute features, like ***opened*** here, are data elements associated with each object of the type.

As a type, a class such as *DEVICE* may be used to declare objects; their features may then be accessed through dot notation, as in:

/15/

```
d1: DEVICE; f1: INTEGER
d1.Create;
d1.open (f1);
if d1.opened then ....
```

Create is a universal procedure applicable to all classes; it allocates the necessary space for an object such as *d1*. If further initialization actions are required, they may be described in a procedure declared in the class with the name *Create*, possibly with parameters.

Note that each routine always has, besides its normal list of arguments, a special argument, the object to which the procedure is applied (*d1* in the above call to *open*). This is one of the characteristics of object-oriented language: every operation is relative to a distinguished object. Within the class, unqualified feature names implicitly refer to this object; the predefined name *Current* may be used when an explicit reference is needed.

These comments account for the "type" aspect of a class. From the "module" standpoint, it should be noted that the class is the only program structuring facility of Eiffel; thus the above example use of *DEVICE* must be in some class, say *C*. A class such as *C* which declares entities (that is to say features, routine parameters or function results) of type *DEVICE* is said to be a client of *DEVICE*. The **export** clause lists the features of a class which are accessible to clients, in read-only mode for attributes and execution mode for routines (here all features shown are exported). Since information hiding is not a concern for this discussion, we shall omit **export...** clauses in the sequel.

The notion of inheritance is a natural extension to this basic framework. Assume we want next to define the notion of tape device. For our purposes, a tape unit has all the properties of devices, as represented by the three features of class *DEVICE*, plus the ability to (say) rewind its tape. Rather than redefining a new class from scratch, we may declare class *TAPE* as an extension of *DEVICE*, as follows:

/16/

```
class TAPE inherit DEVICE feature
   rewind is
      do ...... end
end -- class TAPE
```

With this declaration, objects of type *TAPE* automatically possess (by "inheritance") all the features of *DEVICE* objects, plus their own (here *rewind*). We say that *TAPE* is an heir to *DEVICE*, which is a parent of *TAPE*. The "descendants" of a class are the class itself and the descendants of its heirs; the reverse notion is that of "ancestor".

A class may of course have more than one heir; for example, *DEVICE* could have *DISK* as another heir, with its own specific features (such as direct access read, etc.). In Eiffel, classes may also have more than one parent; this is known as multiple inheritance, a very powerful technique for reusability, allowing the combination of more than one previously developed environment. Eiffel also introduces the technique of "repeated inheritance", making it possible to inherit more than once from the same class.

From the module viewpoint, the ancestor relation is a program structuring mechanism; from the type viewpoint, it yields a rule on acceptable assignments. The rule is simple: an assignment of the form

```
x := y
```

where *x* and *y* are of class types, is only permitted if the type of *x* is a descendant of the type of *y*. Thus the above assignment is legal if, for example, *x* has been declared as a device and *y* as a tape. This may be explained by noting that the inheritance relation is really the "is-a" relation [4]: every tape is a device, but every device is not a tape.

It sometimes happens that a feature of a class should be implemented differently in some descendants of the class. For example there could be a special "open" mechanism for tape devices. Eiffel allows such redefinitions, as follows:

/17/

```
class TAPE inherit
   DEVICE redefine open
feature
   open (file_descriptor: INTEGER) is
      do ..... special open for tape devices .... end;
   rewind is
      do ...... end
end -- class TAPE
```

This possibility must be seen in connection with the above assignment rule: if *x* is a device, then the call

```
x.open (f1)
```

may now be executed differently depending on the assignments that have been performed on *x* before the call is executed: for example, after *x* := *y*, where *y* is a tape, the tape version should be executed. Such feature redefinitions are common in Eiffel programming, which also allows a parameterless function to be redefined as an attribute (which is useful for changing representations in program refinement).

This facility characterizes the powerful brand of polymorphism offered by object-oriented languages with inheritance: the same feature reference may have several interpretations depending on the actual form of the object at run-time. To achieve this effect, many object-oriented languages have renounced static type checking; Eiffel, however, is statically typed (and the binding of feature names to actual features is done statically whenever possible).

The remarkable benefits of the inheritance technique with respect to reusability, extendibility and compatibility come from the fact that software elements such as *DEVICE* are both usable as they are (they may be compiled as part of an executable program) and still amenable to extensions (if used as ancestors of new classes). Thus a compromise between usability and flexibility, fundamental for the qualities mentioned, is achieved.

One more property of Eiffel, borrowed from Simula, will be useful for the discussion below: deferred features (corresponding to Simula's "virtual procedures"). Deferred features correspond to operations that must be provided on all objects of a class, but whose implementation may only be given in particular descendants of the class.

Assume for example that, as under Unix, devices are a special kind of files: *DEVICE* should thus be an heir to class *FILE*, whose other heirs may be *TEXT_FILE* (itself with heirs *NORMAL* and *DIRECTORY*) and *BINARY_FILE*. Figure 1 shows the inheritance graph, a tree in this case.

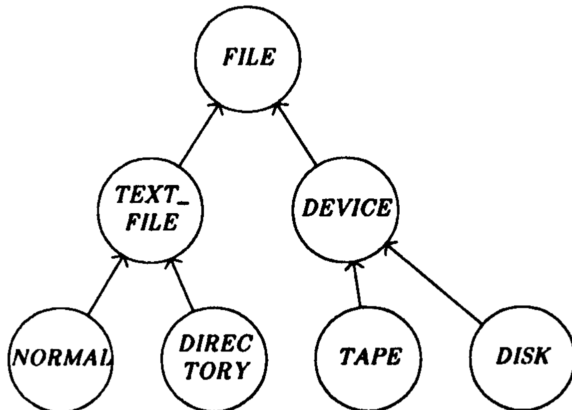

Figure 1: Inheritance graph for files

Any file may be opened or closed; but how these operations are performed depends on whether the file is a device, a directory etc. Thus at the *FILE* level we declare the corresponding procedures as deferred; this means that only a header is given, and that the task of providing an implementation is handed over to descendant classes:

```
/18/
  class FILE feature
     open (file_descriptor: INTEGER) is deferred end;
     close is deferred end;
  end -- class FILE
```

Descendants of *FILE* should provide actual definitions of *open* and *close*. The rules of the language prohibit application of these features to objects for which they might not be defined.

An interesting application of this technique is for Ada or Modula-like separation between interface and implementation of a module: although an Eiffel class is normally defined as a single piece, the effect of Ada's two level declaration (specification and body) may be achieved by declaring a first class with deferred features only, and a second one, heir to the first, with the implementation of these features. An important advantage of this technique over its Ada equivalent is that it allows different implementations of the same feature to coexist in a single software system.

4 - SIMULATING INHERITANCE WITH GENERICITY

To compare genericity with inheritance, we shall study how, if in any way, the effect of each feature may be simulated in a language offering the other.

First consider a language, such as Ada, which offers genericity but not inheritance. Is there any way we can achieve the effects of inheritance in such a language?

The easy part is the overloading. In a language such as Ada or Algol 68 where the same subprogram name may be reused as many times as needed provided it is applied to operands of different types, there is no difficulty in defining types such as *TAPE*, *DISK*, etc., each with its own version of *open*, *close* etc.:

```
/19/
  procedure open
     (p: in out TAPE; descriptor: in INTEGER);
  procedure close (p: in out DISK);
  etc.
```

Provided the subprograms are distinguished by the type of at least one operand, as is the case here, no ambiguity will arise.

However this solution falls short of providing true polymorphic entities as in languages with inheritance, where, as discussed above, an operation will be carried out differently depending on the particular form of an entity at run-time (even though it is possible, in a language like Eiffel, to check at compile time that the operation will be defined in all possible cases). The typical example is the call *d.close*, which will be carried out differently after the assignments *d := di* and *d := ta* (where *di* is a *DISK* and *ta* is a *TAPE*).

The above form of Ada-like overloading does not provide anything like this remarkable possibility.

The only feature of Ada which could be used to emulate this property of object-oriented languages is in fact shared with Pascal and has nothing to do with overloading or genericity: it is the record with variant type. We could for example define something like

```
/20/
  type DEVICE (unit: DEVICE_TYPE) is
     record
        ....... fields common to all device types ........
        case unit is
           when tape => ...... fields for tape devices .....;
           when disk => ...... fields for disk devices .....;
           ....... other cases ......;
        end case
     end record
```

where *DEVICE_TYPE* is an enumeration type with elements *tape*, *disk* etc. Then there would be a single version of each the procedures on devices (*open*, *close* etc.), each containing a case discrimination of the form

```
/21/
  case d'unit is
     when tape => ..... action for tape devices .....;
     when disk => ..... action for disk devices .....;
     ....... other cases ......;
  end case
```

Such a solution, however, is unacceptable from a software engineering point of view: it runs contrary to the criteria of extendibility, reusability and compatibility. Not only does it scatter case discriminations (here on

DEVICE_TYPE) all over the program; worse yet, it **closes** the set of possible choices: as opposed to the Eiffel class *DEVICE* which can at any time be used as parent or a new class, the Ada type *DEVICE* has a fixed list of variants, corresponding to the elements of the fixed enumeration type *DEVICE_TYPE*: to add an element to this list, one must modify the declaration of *DEVICE*, invalidating any program unit that relied on the initial version.

So the answer to the question posed at the beginning of this section — can inheritance be simulated with genericity? — is no.

5 - SIMULATING GENERICITY WITH INHERITANCE

We now address the reverse problem: can we achieve the effect of Ada-style genericity in an object-oriented language with inheritance?

As before, we use Eiffel as our vehicle for expressing object-oriented techniques. As explained in section 6 below, Eiffel does provide a generic parameter mechanism (included in the language as a result of the study reported here); but of course, since the object of this section is to analyze how one may simulate genericity with inheritance, we must temporarily refrain from using the Eiffel generic mechanism. The reader should thus be warned that the solutions presented in this section are substantially more complex than those obtainable with full Eiffel, described in section 6.

The simulation turns out to be easier, or at least less artificial, for constrained genericity — a surprising result since unconstrained genericity is conceptually simpler. Thus we begin with the constrained case.

5.1 - Constrained genericity: overview

The idea is to associate with a constrained formal generic type parameter a class. This is a natural thing to do since a constrained generic type may be viewed, together with its constraining operations, as an abstract data type. Consider for example the generic clauses in our two constrained examples, minimum and matrices:

/22/

```
generic
   type T is private;
   with function "<=" (a, b: T)
      return BOOLEAN is <>;
```

/23/

```
generic
   type T is private;
   zero: T;
   unity: T;
   with function "+" (a, b: T) is <>;
   with function "*" (a, b: T) is <>;
```

We may view the first as a definition of an abstract data type, say *COMPARABLE*, characterized by a comparison operation "<="; similarly, the second specifies a type, say *RING*, with features *zero*, *unity*, "+" and "*".

In an object-oriented language, these types may be directly represented as classes. Such classes may not be entirely specified since there is no general implementation for "<=", "+" etc.; rather, they are to be used as ancestors of actual classes corresponding to actual generic parameters. Here the deferred feature mechanism of Eiffel is exactly what is needed. Thus we define the following classes to represent the generic parameters:

/24/

```
class COMPARABLE feature
   le (other: COMPARABLE): BOOLEAN
      is deferred end
end -- class COMPARABLE
   -- le corresponds to "<=";
   -- there are no infix functions in Eiffel.

class RING feature
   plus (other: RING) is deferred end;
   times (other: RING) is deferred end;
   zero: RING;
   unity: RING
end -- class RING
```

The comment made in section 2.2 about the lack of semantic specification in Ada constrained genericity would seem to apply here too: we have not specified any of the required properties on *le*, *plus* etc. Eiffel does, however, permit the specification of such properties in the form of **preconditions** and **postconditions** on routines. Simple examples of this facility will be given in section 5.4.

The reader will also have noted that *plus* and *times* are defined here as procedures rather than functions; the convention we will follow in the Eiffel examples is that *r.plus* (*r*1) is an instruction that performs a side-effect on *r*, adding to its value the value of *r*1, rather than an expression returning the sum of these values (and similarly for *times*). In contrast, the Ada operators "+" and "*" were functions. The difference is not essential and we use procedures in Eiffel mainly for brevity. The examples may be changed into functions, as in

```
plus (other: RING): RING is deferred end;
```

subject to the discussion that follows.

5.2 - Constrained genericity: subprograms

A subprogram such as *minimum* may now be written by specifying its arguments to be of type *COMPARABLE*. Based on the Ada pattern, the function would be declared as

/25/

```
minimum (one: COMPARABLE; other: COMPARABLE):
      COMPARABLE is
      -- Minimum of one and other
   do ..... end
```

In an object-oriented language, however, every routine (Eiffel term for subprogram) appears in a class and is relative to the "current" object of that class; thus it seems preferable

to include *minimum* in class *COMPARABLE*, argument *one* becoming the implicit current object. The class becomes:

/26/
```
class COMPARABLE feature
   le (other: COMPARABLE): BOOLEAN is deferred end;
   minimum (other: COMPARABLE): COMPARABLE is
         -- Minimum of current element and other
      do
         if le (other) then Result := Current
         else Result := other end
      end -- minimum
end -- class COMPARABLE
```

(The predefined variable *Result* contains the result to be returned by any function in which it appears; it is implicitly declared of the function's result type, here *COMPARABLE*). To compute the minimum of two elements, we must declare them of some descendant type of *COMPARABLE*. For example, we may declare:

/27/
```
class INT_COMPARABLE inherit
   COMPARABLE
feature
   le (other: INT_COMPARABLE): BOOLEAN is
         -- Is current element less than or equal to other?
      do Result := value <= other.value end
   value: INTEGER;
      -- Value associated with current element
   change_value (new: T) is
         -- Make new the value associated
         -- with current element
      do value := new end;
end -- class INT_COMPARABLE
```

To find the minimum of two integers, we may now apply function *minimum*, not to arguments of type integer, but to arguments of type *INT_COMPARABLE*, say *ic1* and *ic2*, as follows:

/28/
```
ic3 := ic1.minimum (ic2)
```

To use the generic *le* and *minimum* functions, we have to renounce direct references to integers, using *INT_COMPARABLE* entities instead; hence the need for attribute *value* and routine *change_value* to access and modify the associated integer values.

We would similarly introduce heirs of *COMPARABLE*, say *STR_COMPARABLE*, *REAL_COMPARABLE*, and so on, for each type for which a version of *minimum* is desired.

Of course, having to declare similar features *value* and *change_value* for all descendants of *COMPARABLE* is unpleasant. But by paying this relatively small price in terms of ease of program writing — renouncing the direct use of predefined types — we achieve the same effect as in a language with genericity.

There is a hitch, however, if we are concerned about static typing. We clearly want to disallow a call such as

/29/
```
ic1.minimum (c)
```

where *c* is a *COMPARABLE* but not an *INT_COMPARABLE*. Function *le* has indeed been redefined to accept only *INT_COMPARABLE* arguments; the rules of Eiffel permit such redefinition of an entity of a class in a descendant of that class, if the new type is itself, as here, a descendant of the original type. But *minimum* has not been redefined; in fact this is the whole point of the game: to make sure that *minimum* is a polymorphic feature, applicable to all kinds of "comparable" objects. So, regrettably, *c* is in fact a legal argument in /29/.

To ensure type consistency we must redefine *minimum* in *INT_COMPARABLE* so that its arguments and result are of type *INT_COMPARABLE*. The body of the routine does not change: only its header has to be modified. The class declaration may thus be rewritten as follows:

/30/
```
class INT_COMPARABLE inherit
   COMPARABLE
      rename minimum as general_minimum;
      redefine minimum
feature
   le (other: INT_COMPARABLE): BOOLEAN is
         ......  As in /27/  .....;
   minimum
      (other: INT_COMPARABLE): INT_COMPARABLE
   is
         -- Minimum of current element and other
      do
         Result := general_minimum (other)
      end; -- minimum
   value: INTEGER;-- As above
   change_value (new: T) is-- As above
      do value := new end;
end -- class INT_COMPARABLE
```

We have used here the renaming mechanism of Eiffel; the **rename...** subclause of the **inherit...** clause makes it possible to access the features of the ancestor class (*COMPARABLE*) even though they are redefined in the descendant. Eiffel prohibits overloading of names within a class, so that renaming is necessary to allow use of both sets of features in the class. (Another use of renaming is in multiple inheritance, to remove name clashes when features are inherited from more than one class).

What we have done is to redefine the header of *minimum*, not its body, which is simply that of the original version, accessible here under the name *general_minimum*. This, apparently, takes care of the static typing conflict to the expense of yet some more complication.

However, the careful reader will have noted that a serious typing problem remains. The call to *general_minimum* is correct with respect to its argument *other*: since *general_minimum* (that is to say,

COMPARABLE's version of *minimum*, as given in /26/) expects *COMPARABLE* objects, an entity like *other* declared of the descendant type *INT_COMPARABLE* is an acceptable substitute under the assignment rule. But there is a problem with the **result** of the function: *general_minimum* returns a *COMPARABLE* whereas *INT_COMPARABLE*'s version of *minimum* should return an *INT_COMPARABLE*.

Thus in the call mentioned in /28/ above, namely

```
ic3 := ic1.minimum (ic2)
```

ic3 should be an *INT_COMPARABLE*; the assignment is illegal if the right-hand side returns just a *COMPARABLE*. In fact, the permitted type combinations in assignments are the inverse ones: the source should be of a descendant type from the target.

With what we have seen so far there is not way to resolve this issue other than by redefining *minimum* completely – not only its header, but its body as well – so that it will indeed return an *INT_COMPARABLE*. This of course defeats the whole purpose of genericity: a similar redefinition must be repeated in each descendant of *COMPARABLE*, with all instances of *minimum* identical except for the type declarations of arguments and results.

We shall only be able to provide a satisfactory solution to this problem by introducing ***declaration by association*** in section 6.

5.3 - Constrained genericity: packages

The previous discussion transposes to packages. We use a class to represent the matrix abstraction implemented in Ada by the *MATRICES* package:

/31/

```
class MATRIX feature
    impl: ARRAY2 [RING];
    entry (i: INTEGER; j: INTEGER): RING is
        -- Value of the (i, j) entry of the matrix
      do Result := impl.entry (i, j) end;
    enter (i: INTEGER; j: INTEGER; v: RING) is
        -- Assign value v to entry (i, j) of the matrix
      do impl.enter (i, j, v) end;
    plus (other: MATRIX) is
        -- Add other to current matrix
      local t1: RING do
        ......... loop .........loop
          t1 := entry (i, j);
          t1.plus (other.entry (i, j));
          enter (i, j, t1 )
        end end end; -- plus
    times (other: MATRIX) is
        -- Multiply current matrix by other
      local ....... do ...... end
end -- class MATRIX
```

Here *ARRAY2* [*T*] denotes a predefined Eiffel class whose elements are two-dimensional arrays of type *T*. Array types are treated in Eiffel as class types; the basic operations on an element *a* of type *ARRAY2* are *a.entry* (*i*, *j*), which returns the *i*, *j* entry of array *a* (that is to say, *a* [*i*, *j*] in standard Pascal notation), and *a.enter* (*i*, *j*, *v*), which assigns value *v* to this entry (that is to say, *a* [*i*, *j*] := *v*). Corresponding operations are declared above for matrices.

We have left out some details (such as how the dimensions of a matrix are set) but outlined the *plus* procedure, exhibiting the object-oriented form of overloading: the internal call to *plus* is the operation on *RING*, not *MATRIX*. Similarly, routines *enter* and *entry* are used in both their *ARRAY2* and *MATRIX* versions.

To define the equivalent of the Ada generic package instantiation (/12/)

```
package BOOL_MATRICES is
    MATRICES (BOOLEAN, false, true, "or", "and");
```

we must declare the "ring" corresponding to booleans:

/32/

```
class BOOL_RING inherit
    RING redefine zero, unity
freeze zero, unity feature
    value: BOOLEAN;
    change_value (b: BOOLEAN) is
        -- Assign value b to current element
      do value := b end;
    plus (other: BOOL_RING) is
        -- Boolean addition: or
      do change_value (value or other.value) end;
    times (other: BOOL_RING) is
        -- Boolean multiplication: and
      do change_value (value and other.value) end;
    zero is
        -- Zero element for boolean addition
      do Result.Create; Result.change_value (false) end;
    unity is
        -- Zero element for boolean multiplication
      do Result.Create; Result.change_value (true) end
end -- class BOOL_RING
```

Note that *zero* and *unity* are redefined as functions returning a value of type *BOOL_RING*. However these are actually constant functions: the clause **freeze...**, not seen before, indicates that *zero* and *unity* are evaluated just once and their values shared among all instances of the class. This is how constants of class types may be introduced in Eiffel.

How do we provide the equivalent to the Ada package instantiation for boolean matrices recalled above? The same reasoning that was applied to class *COMPARABLE* and function *minimum* prevents us from keeping *MATRIX* as it is if type checking is a concern: we want to make sure that an integer element, say, may not be entered into a boolean matrix. To achieve this, we define an heir *BOOL_MATRIX* of *MATRIX*, where routines *entry*, *enter*, *plus* and * are redefined to act only on objects of type *BOOL_RING* rather than any *RING*. As with *minimum*, only the headers of the routines have to be changed, not their implementations; this is achieved as follows, using again renaming to allow access to redefined features of the parent class.

/33/
```
class BOOL_MATRIX
inherit
  MATRIX
    rename entry as general_matrix_entry,
      enter as general_matrix_enter,
      plus as general_matrix_plus,
      times as general_matrix_times;
    redefine impl, entry, enter, plus, times
feature
  impl: ARRAY2 [BOOL_RING]
  entry (i: INTEGER; j: INTEGER): BOOL_RING is
      -- Value of the (i, j) entry of the matrix
    do Result := general_matrix_entry (i, j) end;
  ... and similarly for enter, plus and times ...
end -- class BOOL_MATRIX
```

The reader may note the same problem for the result of function ***entry*** as previously discussed for ***minimum***: this result should be of type *BOOL_RING*, but ***general_matrix_entry*** will only return a *RING*. With the language features seen so far, all we can do is to redefine the body of ***entry***, making it a copy of the body of ***general_matrix_entry*** rather than a call to this routine; then the result will be of the right type. Note that the problem only arises for functions, so the other routines of the class are not affected.

This problem notwithstanding, this construction achieves with inheritance the effect of constrained genericity. This result has been obtained at the price of a certain heaviness in expression; note in particular that what has been done for *BOOL_MATRIX* must be repeated for any descendant of *MATRIX* corresponding to a generic instantiation, e.g. *INT_MATRIX*, *REAL_MATRIX* etc. In addition, features ***value*** and ***change_value*** must be declared anew in each descendant of the associated class *RING*. We shall see in section 6 how such heaviness may be removed.

5.4 - Unconstrained genericity

The mechanism for simulating unconstrained genericity is the same; this case is simply seen as a special form of constrained genericity, with an empty set of constraints. Generic formal type parameters have been interpreted as abstract data types: when unconstrained, they will be seen as abstract data types with no relevant operations. The technique works, but it suffers from the heaviness mentioned above, becoming less tolerable here as the dummy types do not correspond to any obviously relevant data abstraction.

Let us apply the previous technique to both our unconstrained examples, swap and stack, beginning with the latter. We need a class, say *STACKABLE*, describing objects that may be pushed onto and retrieved from a stack. Since this is true of any object, this class has no property of its own beyond its name:

/34/
```
class STACKABLE end
```

We may now declare a class *STACK*, whose operations apply to *STACKABLE* objects:

/35/
```
class STACK feature
  space: ARRAY [STACKABLE];
  index: INTEGER;
  size: INTEGER;
  empty is
      -- Is the stack empty?
    do Result := (index = 0) end;
  push (x: STACKABLE) is
      -- Add x on top of the stack
    require index < size do
      index := index+1;
      space.enter (x)
    end; -- push
  top: STACKABLE is
      -- Last element pushed
    require not empty do
      Result := space.entry (index)
    end; -- pop
  pop is
      -- Remove last element pushed
    require not empty do index := index - 1 end;
  Create (m: INTEGER) is
      -- Create stack with space for m values
    do space.Create (1, m); size := m end
end -- class STACK
```

The **require**... clauses illustrate how routine preconditions (which must be satisfied by actual parameters upon entry to a routine) are written in Eiffel. Postconditions and class invariants may also be expressed (in **ensure**... and **keep**... clauses). This aspect of the language falls beyond the scope of this discussion; see [13] for more details.

STACK relies on the predefined class *ARRAY* for one-dimensional arrays, whose main procedures are ***entry***, ***enter*** and ***Create***; the latter takes two arguments and allocates the array with the values of these arguments as bounds. The ***Create*** procedure for stacks takes just one argument (the stack size).

To instantiate this definition for stacks of specific types, we apply the same techniques as above: define descendants of *STACKABLE*, such as

/36/
```
class INT_STACKABLE inherit STACKABLE feature
  value: INTEGER;
  change_value (n: INTEGER) is
      -- Make n the value of the current element
    do value := n end
end -- INT_STACKABLE
```

and similarly *STR_STACKABLE*, etc.

Here we run again into the typing problem evidenced by ***minimum*** (/30/) and *BOOL_MATRIX* (/33/). Stacks declared simply of type *STACK* cannot be statically guaranteed to contain only objects of a certain class of

"stackables", say *INT_STACKABLE*; and we have the problem of the type of the result returned by function ***top***. In the following sequence

/37/

```
s: STACK; ins: INT_STACKABLE
.........................................
s.Create (10);
ins.Create; ins.change_value (50);
s.push (ins);
ins := s.top
```

the last assignment has a left-hand side of type *INT_STACKABLE* and a right-hand side of type *STACKABLE*: this is typewise wrong even though the code seems quite legitimate semantically (one pushes the value of a variable and retrieves it immediately into the same variable).

For both these reasons, it is necessary to do as in the previous examples, that is to say declare heirs to *STACK*, such as *INT_STACK*, *STR_STACK* etc. Features of *STACK* will be redefined in each of these classes, but only to adapt the types of their arguments and, in the case of ***top***, of the result. Thus for example *INT_STACK* will contain feature redefinitions such as

/38/

```
space: ARRAY [INT_STACKABLE];
push (x: INT_STACKABLE) is
  do general_stack_push (x) end;
```

etc. (the reader may complete this example based on the *MATRIX* case).

The other unconstrained example, procedure ***swap***, may be treated along the same lines; a class *SWAPPABLE* will be introduced. The treatment is left to the reader.

6 - GENERICITY AND INHERITANCE IN EIFFEL

We may draw the following conclusions from the previous discussion.

- Inheritance is the more powerful mechanism. There is no way to provide a reasonable simulation with genericity.
- The equivalent of generic subprograms or packages may be expressed in a language with inheritance, but one does not avoid the need for certain spurious duplications of code. The extra verbosity is particularly hard to justify in the case of unconstrained genericity, for which the simulation mechanism is just as complex as for the conceptually more difficult constrained case.
- Type checking introduces difficulties in the use of inheritance to express generic objects.

To address these issues, Eiffel offers a limited form of genericity and the notion of declaration by association. (The specification language LM, associated with the M specification method, [12], relies on a similar tradeoff).

6.1 - Simple genericity

Since unconstrained genericity is both the simpler case and the one for which the pure inheritance solution is least acceptable, it seems adequate to provide a specific mechanism for this case, distinct from the inheritance mechanism. Consequently, Eiffel classes may have unconstrained generic parameters. A class may be defined as

```
class C [T1, T2, ...., Tn] .....
```

where the parameters represent arbitrary types (simple or class). An actual use of the class will use actual type parameters, as in

```
x: C [INTEGER, RING, ...., DEVICE]
```

We have in fact already encountered such parameterized classes: the basic classes *ARRAY* (section 5.4) and *ARRAY2* (section 5.3) are naturally generic. It should also be noted (although the present paper is about concepts rather than implementation) that Eiffel compilation techniques make it possible to generate a single object module for a parameterized class, as opposed to Ada techniques which treat generic packages as macros to be expanded anew for each instantiation.

The examples of the previous sections provide obvious cases where generic parameters are useful. Take for instance *COMPARABLE* (/26/), which becomes

/39/

```
class COMPARABLE [T] feature
  le (other: COMPARABLE [T]): BOOLEAN is
    deferred
  end;
  minimum (other: COMPARABLE [T]):
    COMPARABLE [T] is
      ... As in /26/ ...;
  value: T;
  change_value (new: T) is do value := new end
end -- class COMPARABLE
```

Here we see an immediate and important benefit of generic parameters: we can solve almost completely the problem of type checking by specifying that the arguments to ***le*** and ***minimum*** and the local variable ***m*** are of type *COMPARABLE* [*T*], for the same *T* as the class itself. Thus we rid ourselves of the necessity to redefine, at least formally, ***minimum*** for each descendant of *COMPARABLE*, which plagued our previous attempts. The generic parameter *T* also allows us to lift the declarations of features ***value*** and ***change_value*** from the various descendants of *COMPARABLE* (see /27/ or /30/) to a single instance in *COMPARABLE* itself.

However we have not yet solved the problem of the type of ***minimum***'s result, which is *COMPARABLE*[*T*] even in a descendant; more on this below.

To define *INT_COMPARABLE* all we have to write now is:

```
/40/
  class INT_COMPARABLE inherit
    COMPARABLE [INTEGER]
  feature
    le (other: INT_COMPARABLE): BOOLEAN is
        -- Is current element less than or equal to other?
      do Result := value <= other.value end
  end -- class INT_COMPARABLE
```

The other examples are treated similarly:

```
/41/
  class RING [T] feature
    plus (other: RING [T]) is deferred end;
    times (other: RING [T]) is deferred end;
    zero: RING [T];
    unity: RING [T];
    value: T;
    change_value (new: T) is do value := new end
  end -- class RING
```

```
/42/
  class MATRIX [T] feature
    impl: ARRAY2 [RING [T]];

    entry (i: INTEGER; j: INTEGER): RING [T] is
      ... As before ... (see /31/);

    ... and similarly for enter, plus and times ...

  end -- class MATRIX
```

Note how the use of a generic parameter in two related classes, *RING* and *MATRIX*, makes it possible to ensure type consistency (all elements of a matrix will be of type *RING* [*T*] for the same *T*). As with *COMPARABLE* (/39/), the declarations of features *value* and *change_value* have been factored out: they now appear in class *MATRIX* rather than being repeated in all its descendants.

In the unconstrained genericity case, the need for dummy classes disappears; class *STACKABLE* and its heirs *INT_STACKABLE*, *STR_STACKABLE* etc. are not needed any more, since *STACK* may be rewritten as

```
/43/
  class STACK [T] feature
    space: ARRAY [T];
    index: INTEGER;
    size: INTEGER;

    ...... The rest of the class as in /35/
    ...... except that T is used in lieu of STACKABLE .......

  end -- class STACK
```

There is no more need for classes such as *INT_STACK*, *STRING_STACK* etc.; simply use *STACK* [*INTEGER*], *STACK* [*STRING*] and so on. The typing problem for ***top*** disappears since the result of this function is now simply of type *T*.

A remarkable degree of simplification has been achieved. Auxiliary classes are not needed any more for unconstrained genericity. However we do ***not*** introduce constrained genericity in the language: this feature would be redundant with the inheritance mechanism. To provide the equivalent of a constrained formal generic parameter, we retain the technique introduced in section 5.1: declare a special class whose features correspond to the constraints (that is to say, the **with** subprograms in Ada terminology), and declare any corresponding actual parameters as descendants of this class. Providing the class with generic parameters simplifies its use and partly solves the type checking problem.

6.2 - Declaration by association

Let us look more closely at the remaining part of the type checking problem. Consider again class *COMPARABLE* as defined last (/39/). Keeping in mind that *COMPARABLE* is intended for use as an ancestor for more specific classes, we do not really want *other* (in both functions), *m* and the result of *minimum* to be of type *COMPARABLE* [*T*]: what is required of these entities is to be of the type of the "current" entity, whatever this may be in a descendant of *COMPARABLE*. When this type changes, we want the other entities to follow suit.

This possibility is achieved in Eiffel through the mechanism of declaration by association. Let a class *C* contain a declaration of the form

```
x: D
```

where *D* is a class type. We may then declare another entity as

```
y: like x
```

Such a declaration means the following: the type of *y* is the same as the type of *x*; if *x* is redefined in a descendant class of *C* as being of a class type *D'*, which must be a descendant of *D*, then *y* will be considered to have been redefined likewise. Note that this is a purely static mechanism; it may be viewed as an abbreviation allowing the redeclaration of just one from a group of related entities to stand for the redeclaration of the whole group.

When the distinguished element of the group, *x* above, is redeclared, it "drags" along all elements declared **like** it. We call it the **anchor** of the association. The anchor may be the current entity, as in

```
y: like Current
```

This readily applies to the previous example:

```
/44/
  class COMPARABLE [T] feature -- Contrast with /39/
    le (other: like Current): BOOLEAN is deferred end;
    minimum (other: like Current): like Current is
      do ... see /26/ ... end;
    value: T;
    change_value (new: T) is do value := new end
  end -- class COMPARABLE
```

Note how this device solves at once all the remaining type checking problems: not only are *le* and *minimum* constrained to act, in all descendants of *COMPARABLE*, on

homogeneous entities (comparing only integers with integers, strings with strings etc.): it also ensures that the result of *minimum* is of the right type, that of its arguments.

The same technique readily applies to the other cases. For example, *RING* (see /41/) becomes:

/45/

```
class RING [T] feature
    plus (other: like Current) is deferred end;
    times (other: like Current) is deferred end;
    zero: like Current;
    unity: like Current;
    value: T;
    change_value (new: T) is do value := new end
end -- class RING
```

In contrast with the *STACK* case, we do need here, because of the deferred procedures, to explicitly declare the descendants of *RING* corresponding to various implementations of *plus* and *times*; for example:

/46/

```
class BOOL_RING inherit
    RING [BOOLEAN]
        redefine zero, unity
freeze
    zero, unity
feature
    ..... as in /32/ ........
end -- class BOOL_RING
```

6.3 - Artificial anchors

For *MATRIX*, a small addition is necessary to ensure that all entities of type *RING* [*T*] are always redefined consistently.

When a group of entities are redefined together by association, one of the entities must serve as the anchor for the association. In the final versions obtained above for *COMPARABLE* and *RING* (/44/ and /45/), the current element is the anchor.

In the *MATRIX* case, the entities to be redefined are of a type different from the current class, namely *RING*. In such a case, there is usually in the current class a feature of the required type which can serve as anchor. For example, the definition of linked lists in the basic Eiffel library [13] uses a class *LINKED_LIST* [*T*] for lists and a class *LINKABLE* [*T*] for list cells, where a list cell contains a value of type *T* and a reference to another list cell. The implementation of a list contains a reference to the first cell of the list; this reference, expressed by a feature *first_element*, is used as anchor for redefinitions of other *LINKABLE* entities of class *LINKED_LIST* in descendants of *LINKED_LIST* (examples of such descendants are the classes defining two-way linked lists and trees, both viewed as special cases of one-way linked lists).

Class *MATRIX*, however, has no feature of type *RING*: the reason is that all "ring" elements are entered into the matrix indirectly, as arguments to procedure *entry*.

Thus we cannot avoid the need to introduce a dummy feature of type *RING* to serve as anchor, as follows.

/47/

```
class MATRIX [T] freeze anchor feature
    anchor: RING [T];
    impl: ARRAY2 [like anchor];
    entry (i: INTEGER; j: INTEGER): like anchor
        is ... As before ... (/31/)...;
    enter (i: INTEGER; j: INTEGER; v: like anchor)
        is ... As before ...;
    plus (other: like Current) is ... As before ...;
    times (other: like Current) is ... As before ...;
end -- class MATRIX
```

(Listing *anchor* in the **freeze** clause avoids the waste of run-time space that would result from physically storing an anchor within each object of the class). Here too specialized classes must be declared for various generic instantiations of *MATRIX*. However, the declarations are now trivial: all that needs to be done is to redefine *anchor*. For example:

/48/

```
class BOOL_MATRIX inherit
    MATRIX [BOOLEAN] redefine anchor
feature
    anchor: BOOL_RING
end -- class BOOL_MATRIX
```

Such a redeclaration closely models the corresponding Ada package instantiation (/12/).

7 - CONCLUSION

Genericity and inheritance are two important techniques towards the software quality goals mentioned at the beginning of this article. We have tried to show which of their features are equivalent, and which are complementary.

Providing a programming language with the full extent of both inheritance and Ada-like genericity would, as we think this discussion has shown, result in a redundant and overly complex design; but including only inheritance would make it too difficult for programmers to handle the simple cases for which unconstrained genericity offers an elegant expression mechanism, like in the stack example.

Thus we have put the borderline at unconstrained genericity. Eiffel classes may have unconstrained generic parameters; constrained generic parameters are treated through inheritance.

Declaration by association completes this architecture by allowing for completely static type checking, while retaining the necessary flexibility.

We hope to have achieved in this design a good balance between the facilities offered by two important but very different techniques for the implementation of extendible, compatible and reusable software.

Acknowledgments

This paper benefited from comments by Vincent Cazala. The work reported was done in part as the author was with the University of California, Santa Barbara.

References

1. Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer, "A Specification Language," in *On the Construction of Programs*, ed. R. McNaughten and R.C. McKeag, Cambridge University Press, 1980.
2. Didier Bert, "Manuel de Référence du Langage LPG, Version 1.2," Rapport R-408, IFIAG, IMAG Institute (Grenoble University), Grenoble, December 1983.
3. Graham Birtwistle, Ole-Johan Dahl, Bjorn Myrhaug, and Kristen Nygaard, *Simula Begin,* Studentliteratur and Auerbach Publishers, 1973.
4. Ronald J. Brachman, "What IS-A and isn't: An Analysis of Taxonomic Links in Semantic Networks," *Computer (IEEE)*, vol. 16, no. 10, pp. 67-73, October 1983.
5. Rod M. Burstall and Joe A. Goguen, "An Informal Introduction to Specifications using Clear," in *The Correctness Problem in Computer Science*, ed. R. S. Boyer and J. S. Moore, pp. 185-213, Springer-Verlag, New York, 1981.
6. Luca Cardelli and Peter Wegner, "On understanding Types, Data Abstraction and Polymorphism," *Computing Surveys (to appear).*
7. Luca Cardelli, "Basic Polymorphic Typechecking," AT&T Bell Laboratories Computing Science Technical Report, 1984, 1986. (Revised version, to appear).
8. Ole-Johan Dahl, Bjørn Myrhaug, and Kristen Nygaard, (Simula) *Common Base Language,* Norsk Regnesentral (Norwegian Computing Center), Oslo, February 1984.
9. K. Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Messeguer, "Principles of OBJ2," in *Proceedings of the 1985 ACM Symposium on Principles of Programming Languages*, vol. 12, pp. 52-66, 1985.
10. Barbara H. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, and Alan Snyder, *CLU Reference Manual,* Springer-Verlag, Berlin-New York, 1981.
11. Bertrand Meyer, "Quelques concepts importants des langages de programmation modernes et leur expression en Simula 67," *Bulletin de la Direction des Etudes et Recherches d'Electricité de France, Série C (Informatique)*, no. 1, pp. 89-150, Clamart (France), 1979. Also in GROPLAN 9, AFCET, 1979
12. Bertrand Meyer, "M: A System Description Method," Technical Report TRCS85-15, University of California, Santa Barbara, Computer Science Department, May 1985.
13. Bertrand Meyer, *Eiffel: a Language for Software Engineering,* Technical Report TRCS85-19, University of California, Santa Barbara, Computer Science Department, November 1985.
14. Robin Milner, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences*, vol. 17, pp. 348-375, 1978.
15. Rishiyur. S. Nikhil, "Practical Polymorphism," in *Functional Programming Languages and Computer Architecture, Nancy (France), 16-19 September 1985, Lecture Notes in Computer Science 201*, ed. Jean-Pierre Jouannaud, pp. 319-333, Springer-Verlag, Berlin-New York, 1985.

Trademarks: Unix (AT&T); Ada (US DoD); Eiffel (Interactive Software Engineering).