Conformance, Genericity, Inheritance and Enhancement

Chris Horn
Distributed Systems Group, Department of Computer Science,
Trinity College Dublin, IRL-Dublin 2 (horn@tcdcs.uucp)

A recent paper by Meyer compared the usefulness of Genericity and Inheritance as a basis for static type checking in object oriented systems. Conformance is also being considered by some researchers, and Meyer's paper is re-examined here using Conformance. Some problems result related to the conformance of formal and actual parameters of operations. Additional rules for conformance are introduced to overcome these problems and these lead naturally to the concept of Enhancement as a union of the techniques of Conformance, Genericity and Inheritance.

This work has been partially sponsored under the ESPRIT programme, in project 834 Comandos - Construction and Management of Distributed Office Systems.

Object-orientation is gaining acceptance as a powerful philosophy for software production. A number of researchers are currently investigating the use of static typing to improve the performance of object-oriented programming languages on conventional hardware architectures, as well as providing early notification of programming errors. In the recent OOPSLA conference, Meyer [Meyer86] informally analysed genericity and inheritance as two techniques to improve the extendibility, re-useablility and compatibility of software subsystems. In particular he noted how genericity could be simulated in a programming language offering only inheritance, but not vice-versa. Nevertheless he concluded that genericity is more elegant than inheritance when the genericity is unconstrained: that is when the actual type parameters substituted for a generic formal type parameter need not support any particular operations. Constrained genericity (in which there is such a requirement on the actual type parameters) can be implemented using inheritance and abstract superclasses.

In addition to genericity and inheritance, conformance and subtyping are being considered by some as the basis for a flexibly typed object based system: for example the Emerald project[Black87], Trellis/Owl [Schaffert86], Galileo [Albano85], as well as [Cardelli85]. Basically, conformance can allow one object (of a particular type) to be used *as if* it were another (of another type). In a system supporting inheritance, (an instance of) a subtype can always be used as if it were (an instance of) any of its supertypes. However conformance may allow a more general relationship between types than that implied by inheritance (whether multiple or single) alone. Whereas inheritance allows one type to extend or redefine the representation of another, one type may conform to another without necessarily obtaining executable code (methods) or data (instance variables) from it. Essentially, an abstract type S conforms to another type T (written $S \leq T$), if [Black86]:

1. S provides at least the operations of T (S may have more operations).
2. For each operation in T, S has the same number of arguments and results.
3. The abstract types of the results of S's operations conform to the abstract types of the results of T's operations.
4. The abstract types of the arguments of T's operations conform to the abstract types of the arguments of S's operations (ie arguments must conform in the *opposite* direction).

Then if $T_1 \leq T_2$:

a. An expression of type T_1 can be assigned to an object of type T_2

b. An actual argument of type T_1 can be assigned to a formal argument of type T_2 in an operation invocation

c. An actual result of type T_2 can be used to receive a formal result of type T_1

Conformance and subtyping are considered more precisely in [Cardelli85]. The chief potentials are the ability to partially order abstract types as well as concrete types [Albano85]; and extensibility and a uniform object model in a distributed environment, by permitting different implementations of the same abstract type [Black87].

In view of the interest in conformance, it is interesting to reconsider Meyer's OOPSLA paper, and determine how the notion of conformance can be successfully integrated with those of genericity and inheritance. In section 1 therefore, Meyer's Swap utility is considered using conformance alone. Problems occur due to the conformance rule 4 above, and in section 2, before continuing to investigate the other problems which Meyer studies, instead an abstract type having multiple implementations is considered, an application for which conformance is reputedly well suited. It consequently becomes apparent that the conformity rules as stated above may be too restrictive, and it is suggested that it is sometimes beneficial to allow one type S to conform to another type T without S necessarily implementing all of T's operations. This introduces the concept of *enhancement*, and in this case we shall say that type S can be *enhanced* to type T. In section 3, nesting of enhanced types is considered. It may sometimes be useful to obtain a new abstract type which is a limited "view" of another - having a subset of the operations - and in section 4 the concept of restriction is introduced as a suitable mechanism, together with the ability to rename operations for convenience. In section 5 Meyer's remaining problems are re-examined using enhancement. Section 6 briefly considers the implications of enhancement on data modelling. Finally in Section 7, we draw some conclusions.

A syntax reminiscent of Modula-2 and Ada is used throughout the following examples, rather than any particular existing object oriented language: the reader is free to alter the syntax where desired.

1 Meyer's swap problem

We start by considering Meyer's first problem, in which an operation is required to swap two instances of some arbitrary type T. Assume there is some type TOP to which all types conform. Using *only* conformity, a utility to swap such instances might be outlined as follows:

```
ABSTRACT TYPE Swapper IS                                              /1/
     Swap(x,y : IN OUT TOP);                  (* x and y are both arguments and results *)
END Swapper;

CONCRETE TYPE SwapperImplementation IS        (* should conform to the Abstract Type *)
     Swap(x,y : IN OUT TOP) IS
          LOCAL t : TOP;
          t := x;  x := y;  y := t;
     END Swap;
END SwapperImplementation;

VAR a,b          : INTEGER;                   (* Let's try swapping two integers....*)
     DoSwap      : SwapperImplementation;
     ....
     DoSwop.Swap(a,b);                        (* this is actually incorrectly typed !!!!! *)
```

Here it is intended that since DoSwop expects a pair of objects conformant to TOP, DoSwop should thus be a polymorphic utility for exchanging any two arbitrarily typed objects. However although the actual arguments a and b of DoSwap.Swap(a,b) conform to the stipulated type of the formal arguments (rule b), the type of the formal results does not conform to that of the actual results (rule c). Hence DoSwop.Swap(a,b) is incorrectly typed. In fact the only objects which can be swapped using DoSwop are those of type TOP! This might have been anticipated, since it might undesireable to swap two objects of *differing* types (eg an INTEGER with a BOOLEAN) as the above example might have permitted. In effect, there is a loss of type information when receiving the DoSwop result parameters.

It would perhaps be more in the spirit of object orientation if the Swap operation were relative to the "current" object: ie that one parameter to Swap is made implicit as the current object. We might anticipate that this does not resolve the problem either:

```
ABSTRACT TYPE Swapper IS                                                  /2/
     Swap(Other : IN TOP) : TOP;                     (* Function returning a result of type TOP *)
END Swapper;

CONCRETE TYPE Swappable IS
     Me : TOP;
     Swap(Other : IN TOP) : TOP IS
          LOCAL t : TOP;
          t := Me; Me := Other; RETURN t;            (* Result returned by function is t *)
     END Swap;
END Swappable;

VAR a,b      : Swappable;
     x       : INTEGER;
     ...
     b := a.Swap(b);                                 (* Type checks OK - but not very useful *)
     x := a.Swap(x);                                 (* Typing error for function result ! *)
```

Instances of type Swappable can indeed be swapped - but that is the only operation they support! More interesting is to try and swap, for example, INTEGERs: however although INTEGER conforms to type TOP, it does not conform to type Swappable - hence we cannot use Swap on INTEGERs.

A concrete type SwappableInteger might be envisaged instead, which would provide a Swap operation for INTEGERs. However if this type had additional operations for INTEGERs - eg a "+" operation - then it would not conform to the abstract type Swappable due to conformance rule 4 above. In any case, it would defeat our original intention to have one implementation of Swap for all types! Meyer notes a similar problem in his paper when using inheritance alone, and introduces his technique of *declaration by association* as a solution.

The conclusion is that genericity, as Meyer advocates, is the most elegant solution to the Swap problem. Although here we have considered just conformance alone (as a possible alternative to genericity), the Emerald and Trellis/Owl projects which do use conformance, are both also including support for genericity.

2 The Sequence

Before considering Meyer's remaining problems, in view of the difficulties encountered above with the Swap problem, let us investigate instead a problem for which conformance is reputedly well suited - that of multiple implementations of some abstract type. A sequence is chosen as a representative problem, to which items can be appended and removed at either the front or tail of the sequence, together with an operation to append two sequences. We first consider only sequences of INTEGERs, and later in section 3 extend to generic sequences:

```
ABSTRACT TYPE IntSequence IS                                              /3/
     Empty                          : BOOLEAN;
     AddFront(i : INTEGER)          : IntSequence;
     RemoveFront                    : INTEGER;
     AddLast(i : INTEGER)           : IntSequence;
     RemoveLast                     : INTEGER;
     Append(s : IntSequence)        : IntSequence;
END IntSequence;
```

Now a supposed implementation which should conform with IntSequence, and which uses an ARRAY:

```
CONCRETE TYPE IntArraySequence IS                                         /4/
     d : ARRAY[1..100] OF INTEGER;
     front,back : CARDINAL INITIALLY 0;

     Empty : BOOLEAN                          ...Implementation of Empty;
     AddFront(i : INTEGER) : IntArraySequence ...Implementation of AddFront;
     RemoveFront : INTEGER                    ...Implementation of RemoveFront;
     AddLast(i : INTEGER) : IntArraySequence  ...Implementation of AddLast;
     RemoveLast : INTEGER                     ...Implementation of RemoveLast;
```

```
        MaxLengthOfSequence : INTEGER          ...Implementation of MaxLengthOfSequence

        Append(s : IntArraySequence) IS
            LOCAL i : INTEGER;
            i := s.front;
            WHILE i <> s.back
                DO back := (back+1) MOD 100;  d[back] := s.d[i];   i := (i+1) MOD 100;  END;
            s.front := s.back;                                                (* s now empty *)
            RETURN SELF;
        END Append;
    END IntArraySequence;
```

For Append, it has been assumed that the representation of another instance of IntArraySequence can be accessed directly. In C++ [Stroustrup 84], this is indeed legitimate; in Smalltalk-80 it is not and one can never directly access the representation of any object other than the current object.

If IntArraySequence is to be an implementation of the abstract type IntSequence, then IntArraySequence must conform to IntSequence. This is it does if IntSequence in turn conforms to IntArraySequence (by virtue of the formal argument of Append, and conformance rule 4). Hence IntArraySequence and IntSequence must be mutually conformant. This is unfortunate since IntArraySequence may have additional operations to IntSequence - such as MaxLengthOfSequence.

To solve this paradox, let us redefine Append in IntArraySequence (/4/) to use the abstract type rather than the concrete type as the formal parameter. However then Append must be coded so that it can append any implementation of IntSequence to an instance of IntArraySequence - that is we cannot directly access the representation of the formal parameter "s" and must use operations upon it (as would be done in Smalltalk-80):

```
        Append(s : IntSequence) IS                                                    /5/
            WHILE NOT s.Empty DO
                SELF.AddLast(s.RemoveFront); END;
            RETURN SELF;
        END Append;
```

IntArraySequence now conforms to IntSequence and the requirement for mutual conformance is relaxed. Naturally now a second and alternative implementation of IntSequence could be given, which would conform to IntSequence and which would use a different representation than IntArraySequence - for example a linked list in a concrete type IntLinkedListSequence. An Append operation in either concrete type would be able to append two instances having different representations - for example a sequence implemented using a linked list could successfully be appended to a sequence using an array. Exactly the same implementation of Append would however appear in *both* concrete types! This would be unexpected, unusual and undesireable in the object approach, in which code is normally shared and reused as much as possible.

Our conclusion is that those operations which require formal parameters of the abstract type itself could sometimes be implemented in conjunction with the abstract type. Using a slightly different syntax:

```
    TYPE IntSequence(R AS IntBasicSequence) IS                                        /6/

        <R>;                                          (* An IntSequence has as representation any
                                                       type which conforms to IntBasicSequence *)
        Append(s : R) : R IS
            WHILE NOT s.Empty
                DO SELF.AddLast(s.RemoveFront); END;
            RETURN SELF;
        END Append;
    END IntSequence;

    TYPE IntBasicSequence IS                          (* An Abstract Type *)
        Empty                    : BOOLEAN;
        AddFront(i : INTEGER)    : IntBasicSequence;
        RemoveFront              : INTEGER;
        AddLast(i : INTEGER)     : IntBasicSequence;
        RemoveLast               : INTEGER;
    END IntBasicSequence;
```

The AS clause specifies that IntSequence is an *enhancive type* - it enhances any type which is conformant with IntBasicSequence. The Append operation takes an instance of the same type as is used for the representation of SELF. An implementation of IntBasicSequence could be:

```
TYPE IntArraySequence IS                                                          /7/
    d : ARRAY[1..100] OF INTEGER;
    front,back : CARDINAL INITIALLY 0;

    Empty : BOOLEAN                                ...Implementation of Empty;
    AddFront(i : INTEGER) : IntArraySequence       ...Implementation of AddFront;
    RemoveFront : INTEGER                          ...Implementation of RemoveFront;
    AddLast(i : INTEGER) : IntArraySequence        ...Implementation of AddLast;
    RemoveLast : INTEGER                           ...Implementation of RemoveLast;
    MaxLengthOfSequence : INTEGER                  ...Implementation of MaxLengthOfSequence;
END IntArraySequence;
```

IntArraySequence is a concrete type. Since IntArraySequence conforms to IntBasicSequence, IntArraySequence can be used as an actual type parameter for the formal type R in /6/. Naturally IntLinkedListSequence could be similarly defined.

As a result, neither of the concrete types IntArraySequence or IntLinkedListSequence need implement Append, and yet can still conform to IntSequence if they implement Empty, AddFront, RemoveFront, AddLast and RemoveLast. So a type T_1 which is to conform to another type T_2 need *cnly conform to certain* operations of T_2 and if it does, will be *enhanced* by further operations. This contrasts with the (conventional) conformance rules 1-4 given earlier.

In a sense we have provided a mechanism similar to abstract superclasses in Smalltalk-80. However the mechanism here is more powerful as a result of using conformance rather than (single) inheritance. It is possible that a given type T may conform to many different types $T_1..T_n$, and using our mechanism, gain access to additional operations in each of these types $T_1..T_n$ which T itself does not implement. Hence the mechanism is akin to inheritance from multiple abstract superclasses, rather than just one. Further, instances of superclasses cannot be created in Smalltalk-80, and finally, the precise "abstract superclasses" in the mechanism here need not be explicitly enumerated at the time the "subclass" is defined. Continuing, an example of usage would be:

```
VAR a : IntSequence;                                                              /8/
    b, c : IntArraySequence;
    i : INTEGER;
    ...
    a := b;                          (* IntArraySequence does conform to IntSequence *)
    a.PutFront(i);
    a.Append(c);                     (* Appends two Array sequences *)
```

(note we allow the result of a function to be ignored if desired) Here "b" as an IntArraySequence is enhanced to an IntSequence, and as a consequence gains an additional operation Append not originally defined for IntArraySequences.

Before examining instantiation of these types the four conformity rules given earlier should first be extended with rules for enhancement, and for expansion of enhancive types.

2.1 Expansion and Enhancement rules

An enhancive type T with n formal type parameters can be *expanded* as follows:

5. If $T(t_1 \text{ AS } T_1, t_2 \text{ AS } T_2,...t_n \text{ AS } T_n)$ and if $p_i \leq T_i$ for all i:1..n then:

5.1 $T = T(T_1,T_2,...T_n)$ ie if T is unqualified it can be treated as $T(T_1,T_2,...T_n)$

5.2 $T(p_1,p_2,...p_r) = T(p_1,p_2,...p_r,T_{r+1},...T_n)$ where $r \leq n$

Secondly, the rules for enhancement:

6. If $T(t_1 \text{ AS } T_1, t_2 \text{ AS } T_2, ... t_n \text{ AS } T_n)$ has representation t_i, where $1 \le i \le n$ - ie

```
TYPE T(t1 AS T1, t2 AS T2,...tn AS Tn) IS
    <ti>;
    .....
END T
```

and if $p \le T_1$, then p --> T(p) (p can be enhanced to T(p))

Enhancement can then be used as follows:

d. **Static usage:** If p --> T(p), then an instance of T(p) can be declared.

e. **Dynamic usage:** If p --> T(p), then an expression of type p can be assigned to an object of type T, or passed as an actual argument in an operation invocation for a formal argument of type T, or returned as a result for an operation invocation where the actual result parameter is of type T.

f. **Temporary usage:** If p -->T(p), then an instance x of type p can be temporarily enhanced to be of type T(p) for a single operation invocation x.T(p)~a, where a is the name of an operation in the signature of T.

g. **Protracted usage:** If p --> T(p), then an instance x of type p can be enhanced protractedly to be of type T(p) using: WITH x AS T(p) DO ... END;

The usage of IntSequence (as defined in /6/), IntBasicSequence (/6/), IntArraySequence (/7/) and IntLinkedListSequence can now be considered. (In fact, we only consider here enhancive types having a single formal type parameter, for simplicity. Multiple formal type parameters requires further rules to resolve possible name clashes) First, assume the following declarations:

```
VAR IntSeq          : IntSequence;                                                      /9/
    IntSeqASeq      : IntSequence(IntArraySequence);
    IntASeq         : IntArraySequence;
    IntLLSeq        : IntLinkedListSequence;        (* NOT conformant with IntArraySeq *)
    i               : INTEGER;
```

(note that we also assume IntArraySequence and IntLinkedListSequence are <u>not</u> mutually conformant). As indicated in rules d to g above, there are four possible different enhancement methods. The first is *Static:* instances of an enhancive type such as IntSequence statically define their representation. When an instance of an enhancive type is declared, actual type parameters can be supplied (for at least some of) the formal types specified in the AS clause of the enhancive type - examples are IntSeq and IntSeqASeq above. Example usage might be:

```
IntSeq.AddFront(45);                 (* 5.2, 4 and b *)             /10/
i := IntSeq.RemoveLast;              (* 5.2, 3 and c *)
IntSeq.Append(IntLLSeq);             (* 5.2, 1-4 and b *)
IntSeqASeq.Append(IntLLSeq);         (* Type error!!! 1-4 and b *)
```

Since the declaration of IntSeq did not define an actual type parameter for its formal parameter in the AS clause, any type conformant with IntBasicSequence can be used subsequently for the parameter. However the same is not true for IntSeqASeq, and the parameter to an Append operation on it must conform to IntArraySequence.

The second method is *Dynamic:* an instance of an enhancive type may change its representation at execution time:

```
IntSeq := IntASeq;                  (* IntSeq is now represented by an array: 5.1, 1-4, 6 and e *)  /11/
IntSeq.Append(IntLLSeq);            (* 5.1, 6 and b *)
IntLLSeq := IntSeq.Append(IntLLSeq); (* Type error!!! 5.1, 6 and c *)
IntSeq := IntLLSeq;                 (* Now change IntSeq's representation: 5.1, 1-4, 6 and e *)
IntSeq.Append(IntASeq);             (* 5.1, 6 and b *)
```

Dynamic usage may also be used for argument transmission and result reception in operation invocations.

The third method is *Temporary:* this allows a concrete type such as IntArraySequence to be enhanced for a single operation invocation:

```
        IntASeq.IntSequence(IntArraySequence).Append((IntLLSeq);          (* 5.2, 6, h and b *)      /12/
```

This is useful to allow a type to temporarily gain access to additional operations which were not originally defined for itself.

The final method is *Protracted:* a concrete type may be enhanced over several statements:

```
        WITH IntASeq AS IntSequence(IntArraySequence)                     (* 5.2, 6 and i *)         /13/
          DO   ....
               IntASeq. Append(IntLLSeq);                                 (* b *)
               ....
        END;
```

3 The Generic Sequence

IntSequence (/6/), IntBasicSequence (/6/) and IntArraySequence (/7/) can now be changed to handle sequences of arbitrary type:

```
      TYPE Sequence(R, AS BasicSequence) IS                                                          /14/
          <R>;

          Append(s : R) : R IS
              WHILE NOT s.Empty
                  DO SELF.AddLast(s.RemoveFront); END;
              RETURN SELF;
          END Append;

      END Sequence;

      TYPE BasicSequence(T as TOP) IS
          Empty                   : BOOLEAN;
          AddFront(i : T)         : BasicSequence;
          RemoveFront                            : T;
          AddLast(i : T)          : BasicSequence;
          RemoveLast              : T;
      END BasicSequence;

      TYPE ArraySequence(T AS TOP) IS
          d : ARRAY[1..100] OF T;
          front, back  : CARDINAL INITIALLY 0;

          Empty : BOOLEAN                       ...Implementation of Empty;
          AddFront(i : T) : BasicSequence       ...Implementation of AddFront;
          RemoveFront : T                       ...Implementation of RemoveFront;
          AddLast(i : T) : BasicSequence        ...Implementation of AddLast;
          RemoveLast : T                        ...Implementation of RemoveLast;
          MaxLengthOfSequence : INTEGER         ...Implementation of MaxLengthOfSequence;
      END ArraySequence;
```

Depending on how instances of these types are declared, homomorphic sequences can be created (in which all items in the sequence have the same type), as well as polymorphic sequences (in which items of differing types can simultaneously be stored in the same sequence). However only instances of TOP can be retrieved from a polymorphic sequence if static type checking is used.

Further rules are required to allow nested enhancive types: those whose formal parameters are instantiated by further enhancive types. For example Sequence has a formal type parameter BasicSequence, which in turn has a formal type parameter TOP.

3.1 Further enhancement and expansion rules for Enhancive Types

7. If $T(t_1 \text{ AS } T_1)$, and $p(t \text{ AS } T_2) \leq T_1(t_2 \text{ AS } T_2)$ and $q \leq T_2$ then:

7.1 $p \dashrightarrow T(p)$

7.2 $p(q) \dashrightarrow T(p(q))$

7.3 $p(q) \rightarrow T(T_1(q))$

7.4 $p(q) \rightarrow T(T_1(T_2))$

Rule 7 generalises to an arbitrary degree of nesting and arbitrary number of parameters. For expansion of nested enhancive types:

8. If $T(t_1$ AS $T_1)$, $T_1(t_2$ AS $T_2)$, and $p \leq T_1$, $q \leq T_2$ then:

8.1 $T = T(T_1(T_2))$

8.2 $T(p) = T(p(T_2))$

Rule 8 likewise generalises.

3.2 Usage of the generic sequence

Examples of the application of rules 7 and 8 are now given:

```
VAR SqASq       : Sequence(ArraySequence);                                                      /15/
    SqASqInt    : Sequence(ArraySequence(INTEGER));
    SqBSqInt    : Sequence(BasicSequence(INTEGER));
    ASq         : ArraySequence;
    ASqInt      : ArraySequence(INTEGER);
    LnkSq       : LinkedListSequence;
    Sq          : Sequence;
    i           : INTEGER;
    t           : TOP;
    ....
    SqASq.AddFront(i);                                   (* 8.2, 5.1 and b *)
    i := SqASq.RemoveFront;                              (* Type error !! 8.2, 5.1 and c *)
    t := SqASq.RemoveFront;                              (* 8.2, 5.1 and c *)
    SqASqInt.AddFront(i);                                (* b *)
    SqASq.Append(LnkSq);                                 (* Type error!! 8.2, 5.1 and b *)
    SqBSqInt := ASqInt;                                  (* 7.3 and e *)
    Sq := ASqInt;                                        (* 8.1, 5.1, 7.4 and e *)
    SqBSqInt.Append(LnkSq);                              (* Type error!! 5.1 and b *)
    Sq.Append(LnkSq);                                    (* 8.1 and 5.1 *)
    ASq := Sq.Append(LnkSq);                             (* Type error! ASq NOT ≤ BSq, & rule c fails *)
    ASq.Sequence(ArraySequence)~Append(LnkSq);           (* 7.1, 5.1 and f *)
    WITH ASq, LnkSq AS Sequence                          (*  5.1 and g *)
        DO  ASq.AddFront(3);                             (* b *)
            t := LnkSq.RemoveFront;                      (* c *)
            ASq.Append(LnkSq);                           (* b *)
    END;
```

4 Restriction and Renaming

Any implementation of the BasicSequence in /14/ could be used to implement abstractions other than that of a Sequence, for example a Stack. An abstract type similar to Sequence in /14/ could be defined for a Stack, but not all the operations applicable to BasicSequence should be allowable on a Stack. Hence although the representation of a Stack could be a BasicSequence, the set of operations made available must be restricted, as follows:

```
TYPE Stack(R AS BasicSequence({AddFront, RemoveFront, Empty}) IS                          /16/
        <R>;
END Stack;
```

where the {..} notation denotes a set of operation names. Any type conformant with BasicSequence may be used for Stack, but only the AddFront, RemoveFront and Empty operations will be invokable on instances of Stack. This obviates the need to introduce an additional type definition explicitly: instead a new type is implicitly introduced having a "hidden" type name and only the specified operations.

It would actually be preferable if the operations on a Stack could be given more meaningful names, and of course, there might also be additional operations on a Stack which are not available on BasicSequences (as there was the operation Append for Sequence), as follows:

```
TYPE Stack(R AS BasicSequence({Push <= AddFront, Pop <= RemoveFront, IsEmpty <= Empty}) IS          /17/
    <R>;

    Empty IS      (* Collapse entire Stack *)
        LOCAL t : TOP;
        WHILE NOT SELF.IsEmpty
            DO t := SELF.Pop;  END;
    End Empty;

END Stack;
```

Here AddFront is renamed as Push, and RemoveFront as Pop, to "users" of a Stack. The renaming implies the first conformance rule (rule 1) must take account of explicit name changes to operations.

As a second example, BasicSequence could be used to implement a FIFO queue:

```
TYPE FIFOQueue(R, R1 AS BasicSequence({Put <= AddFront, Get <= RemoveLast, Empty}) IS          /18/
    <R>;
END FIFOQueue;
```

Note that now we can treat the *same* instance of an ArraySequence (/14/) as a Sequence (/14/), or as a Stack (/17/), or as a FIFOQueue (/18/)!

```
VAR a1,a2 : ArraySequence;                                                                       /19/
    b     : Sequence;
    c     : Stack;
    d     : FIFOQueue;
    ...
    b := a1;                              (* Treat a1 as a Sequence *)
    b.Append(a2);                         (* a1 can use the "inherited" Append  *)
    c := a1;                              (* Now treat a1 as a Stack *)
    c.Empty;                              (* a1 can be flushed *)
    d := a1;                              (* Finally treat a1 as a FIFOQueue *)
    d.Put(90);                            (* a1 supports Put as a synonym for AddFront *)
```

4.1 Restriction and Enhancive Types

Although BasicSequence does conform with BasicSequence({AddFront, RemoveFront, Empty}) as in /17/, in general a type T does not necessarily conform with a restriction of itself. T shall be called the *base* type of the restriction. For example, INTEGER does not conform to INTEGER({"+"}), because of conformance rule 4 and because the arguments to "+" for the restricted type will themselves be of type INTEGER({"+"}). With restricted types, we can introduce a further rule:

9. If an enhancive type is given an actual type parameter which is a restricted type, then any actual arguments or results of operations, whose formal types are the restricted type, can be any subtype of the base of the restricted type.

Examples of the use of this rule are given below when constrained genericity is considered.

5 Constrained Genericity

Having considered how to extend the conventional conformance rules, let us now return to Meyer's problems. Using the additional rules 5 to 8 given above, it is straight forward to describe solutions to both the Swap utility and the Stack which Meyer considers: this is left as an exercise for the interested reader. His examples of constrained genericity require both the additional rules and the restriction and renaming mechanism introduced in section 4, as follows.

Meyer's first example of constrained genericity is a utility to find the minimum to two values of the same arbitrary type: however their type must obviously support a comparison operation. Rule 9 above allows a solution since an INTEGER can be passed as an actual parameter for a formal parameter of type INTEGER({le <= "+"}), even though INTEGER does not of itself conform to the restriction. It is also left as an exercise to the reader.

The second of Meyer's examples of constrained genericity is a matrix package, which allows matrices whose elements are of the same type to be added and multiplied: this type must support add and multiply operations itself (on the elements) as well as a zero and unity value. Such a type is a Ring:

```
TYPE Matrix(R AS TwoDimensionalArray(RingType)) IS                                    /20/
    <R>;                                              (* a 2-D array of RingType *)

    Plus(Other : R) : R IS              ...Implementation of Plus;
    Multipy(m : R) : R IS               ...Implementation of Multiply;

END Matrix;

TYPE RingType IS
    Zero      : RingType;
    Unity     : RingType;
    Plus(Other : RingType) : RingType;
    Multiply(Other : RingType) : RingType;
END RingType;
```

and a possible usage:

```
VAR m1,m2 : Matrix(TwoDimensionalArray(INTEGER({Zero <= 0,Unity <= 1, Plus <= "+", Multiply <= "*"}));     /21/
    ...
    m1.Plus(m2);
```

6 Data Modelling

So far an approach based on abstract data types has been used in the discussion and examples. This resulted from our consideration of conformance in Emerald, and Meyer's particular examples. However conformance between one type and another can also be based on data valued attributes: for example, a record (or tuple) type R_1 would conform to another record type R_2 if R_1 had at least the fields of R_2, and the types of those fields conform. [Albano85] and [Cardelli85] consider subtyping and conformance in greater detail.

The examples have shown how a type could be enhanced by additional operations not originally defined upon it. In the same way, a record type may in principle be augmented by additional fields, and thus enhanced using static, dynamic, temporary or protracted enhancement. Dynamic enhancement appears particularly useful since it allows an object to gain additional attributes, and adopt many different roles, as its "life" progresses, as in /19/. Exactly what these additional attributes are need not be decided when the object is first created, but can be added subsequently. It appears possible to enhance the object and modify its behaviour throughout its lifetime, yet without abandoning static type checking.

Note also that given N enhancive types, 2^N combinations of types can be created. This mechanism is a solution to the addition of *mix-ins* to object instances, as discussed by Hendler [Hendler86].

7 Conclusions

We have observed that it may be useful to extend the usual rules for conformance so that a type can be enhanced to gain additional operations or data attributes, and yet still be statically type checked. In a sense the extension is similar to multiple abstract superclassing. However the major difference from languages which do support multiple inheritance is that using enhancement, the "inheritance" need not be statically determined, but can be achieved at execution time.

In retrospect, the concept of enhancement given here is similar to the *descriptive classes* of Sandberg [Sandberg86], although enhancement is derived from considering conformance, and descriptive classes from abstract superclassing. The chief difference appears that when an instance of a descriptive class is created, actual class parameters must be supplied for all the formal class parameters. As a result such an instance cannot augment its representation at run-time, and for example, much of example /15/ given here might not be possible using descriptive classes.

The concept of enhancement is now being considered as a modelling and implementation tool for the infrastructure for Office Systems, and other application environments, in the context of the ESPRIT Comandos project.

8 Acknowledgements

I am indebted to my colleagues on the Comandos project at Trinity for their constructutive criticisms: V. Cahill, A. Donnelly, E. Finn, N. Harris and I. White. I would also thank my colleagues in the other institutions participating in Comandos, and in particular R. Balter, E. Bertino, R. Gagliardi, L. Martino, G. Mueller and A-K Proefrock.

9 References

[Albano85] "Galileo: A Strongly-Typed, Interactive Conceptual Language", A. Albano, L. Cardelli and R. Orsini, ACM Transactions on Database Systems, Vol. 10, No. 2, June 1985

[Black86] "Object Structure in the Emerald System", A. Black, N. Hutchinson, E. Jul and H. Levy, Technical Report 86-04-03, Department of Computer Science, University of Washington, April 1986.

[Black87] "Distribution and Abstract Types in Emerald", A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter, IEEE Transactions on Software Engineering, Vol SE-13, No 1, January 1987.

[Cardelli85] "On understanding Types, Data Abstraction, and Polymorphism", L. Cardelli and P. Wegner, ACM Computing Surveys, Vol 17, No 4, Dec 85.

[Goldberg83] "SmallTalk-80: The Language and Its Implementation", A. Goldberg and D. Robson, Addison-Wesley, 1983

[Hendler86] "Enhancement for multiple-inheritance", J. Hendler, ACM Sigplan Notices, Vol 21, No 10, October 1986.

[Meyer86] "Genericity versus Inheritance", B. Meyer, 1986 Proc. of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) (also in ACM SIGPLAN Notices, Vol 21, No 11, November 1986)

[Sandberg86] "An Alternative to Subclassing", D. Sandberg, 1986 Proc. of Object-Oriented Programming Systems, Languages and Applications (OOPSLA), (also in ACM SIGPLAN Notices, Vol 21, No 11, November 1986).

[Schaffert86] "An introduction to Trellis/Owl", C. Schaffert et al, 1986 Proc. of Object- Oriented Programming Systems, Languages and Applications (OOPSLA) (also in ACM SIGPLAN Notices, Vol 21, No 11, November 1986).

[Stroustrup84] "The C++ Programming Language - Reference Manual", B. Stroustrup, AT&T Bell Labs Computing Science Technical Report No 108, January 1984.