A Proposal for Making Eiffel Type-safe

W. R. COOK

Hewlett-Packard Laboratories, 1501 Page Mill Road, P.O. Box 10490, Palo Alto, CA 94303-0969, USA

Statically type-correct Eiffel programs may produce run-time errors because (1) attributes may be redeclared during inheritance, invalidating assignments in the superclass, (2) a formal method argument type may be restricted in violation of the contravariance of function types, and (3) two applications of a generic class are assumed to conform if the actual arguments conform. The third problem is solved by case analysis on the variance of generic parameters. Declaration by association provides a solution to the first two problems, but it suffers from additional difficulties.* Type attributes, *or generic parameters with default values, are suggested as a replacement for most cases of declaration by association. The special association type used to express type recursion cannot be explained using type attributes, and it appears to be a truly novel construct for typing object-oriented programs. One consequence of this construct is that Eiffel's conformance hierarchy is a proper subset of its inheritance hierarchy.

Received April 1989

1. INTRODUCTION

This paper discusses several problems in the Eiffel type system, as defined in the book *Object-Oriented Software Construction*.[6] We argue that the language has three fundamental errors connected to the contravariance of function types. In particular, the Eiffel type system seems to assume that a function with a restricted set of legal inputs can be used in a context where a function with a wider set of legal arguments is expected – with the obvious consequence that illegal values may be passed to functions during execution. This assumption causes problems in the conformance rules for type redefinition and generic classes. The obvious approach to fixing the problems would limit the expressive power of the language. To avoid this, Eiffel's novel mechanism of declaration by association is examined as a way to express similar, type-safe programs. However, further problems in declaration by association are identified, which in the end call for its complete reformulation in terms of explicit *type attributes*. The special case of using declaration by association to express recursive types cannot be handled by type attributes, and is identified as a truly novel construct for typing object-oriented programs. We also discuss problems in using the export mechanism for encapsulation.

The next section of this paper is a short review of the typing rules in the definition of the language. Sections 3 and 4 demonstrate problems caused by redeclaration of attributes and methods respectively, and propose corrected typing rules. Section 5 shows how these problems affect formal argument types declared by association, and argues that a restriction of the conformance relation is needed. Section 6 discusses the problem of conformance of generic class types. Section 7 describes some problems with declaration by association, and suggests that they be reformulated as explicit type attributes and treated like generic parameters. Section 8 illustrates the inability of the export clause to ensure encapsulation. All of the Eiffel code given has been compiled and type-checked without errors using the Eiffel system, and they all signal fatal exceptions when run (except the last example, which merely violates encapsulation). The conclusion summarises our recommendations for Eiffel.

2. THE EIFFEL TYPE SYSTEM

Eiffel is a strongly typed object-oriented programming language with a rich type system and a conformance (subtype) relation linked to the inheritance hierarchy. A type in Eiffel is either a simple type (*REAL*, *INTEGER*, etc.), a class type $P[U_1, \ldots, U_n]$ where P is a generic class with n parameters and $U_1, \ldots, U_n$ are types, a formal parameter T inside a parameterised class definition, or an association type of the form `like` *anchor*. When a class type does not have generic parameters (i.e. $n = 0$) it is written as P (instead of $P[\,]$).

A class definition specifies the *features* of the class. Features may be either *attributes* or *routines* (we will use the more familiar term *method* instead of *routine*). The attributes define the local state of each instance of the class (attributes are called *instance variables* in most oject-oriented languages). Methods define the behaviour of instances. A class must export a feature in order for it to be used outside the class definition. The special symbol `Current` plays the role of `self` in Smalltalk or `this` in C++. The value of an attribute can only be changed from within the class in which it is defined, although it can be accessed from outside. Inheritance is used to define a new class as an extension/modification of an existing class.

The two Eiffel classes presented below serve as the basis for later examples. The class `Base` has a single method `base` which multiplies its argument by two. The subclass `Extra` has an additional method `extra` that squares its argument. The *only* significant aspect of these classes is that `Extra` has a feature not present in `Base`. Export clauses have been omitted for brevity; we assume that all features are exported.

```
class Base feature
  base(n : Integer) : Integer is
    do Result := n * 2 end;
end

class Extra inherit Base feature
  extra(n : Integer) : Integer is
    do Result := n * n end;
end
```

In Eiffel, *type conformance* is a relation intended to capture the notion of one type being immediately compatible with another, in the sense that in a context where a value of some type is expected, any value of a conforming type can be used. Conformance is often called *subtyping*. In Eiffel, conformance follows the inheritance hierarchy: Y conforms to X if Y inherits from X. This is the definition of conformance in Eiffel.[6]

> A type Y is said to conform to a type X if and only if one of the following applies:
> (1) X and Y are identical;
> (2) X and Y are class types, X has no generic parameters, and Y lists X in its inheritance clause;
> (3) X and Y are class types, X is of the form $P[U_1, U_2, \ldots, U_n]$, and the inheritance clause of Y lists $P[V_1, V_2, \ldots, V_n]$ as parent, where every V_i conforms to the corresponding U_i;
> (4) Y is of the form `like` *anchor*, and the type of *anchor* conforms to X;
> (5) there is a type Z such that Y conforms to Z and Z conforms to X;

The link between conformance and inheritance is indicated by the use of the phrase 'Y lists X in its inheritance clause'. In the example given above, the type `Extra` conforms to the type `Base`.

Since conformance is defined directly in terms of inheritance, the inheritance mechanism should guarantee that subclass instances can in fact be used wherever a parent instance is expected. This is ensured by restrictions on the kinds of modification that can be made during inheritance. Besides the traditional modifications of adding attributes or changing the implementation of methods, Eiffel allows the types of features to be changed. The redeclaration of features during inheritance is governed by a *type redefinition ruel.*[6]

> An attribute, a function result or a formal routine argument declared in a class may be redeclared with a new type in a descendant class, provided the new type conforms to the original one.

This rule is intended to ensure that instances with redeclared attributes will in fact function properly when used where instances of the superclass are expected.

Type compatibility is an extension of type conformance that determines when values of one type may be assigned to variables of another type or passed as parameters to a procedure expecting another type of value. Type compatibility is essentially the same as type conformance, except that it defines the primitive type *INTEGER* to be compatible with *REAL*; this special compatibility seems to be separated from the general conformance rule because it requires a coercion, while conformance in general does not. In the example above, `Extra` is type-compatible with `Base`.

Types of the form `like` *anchor* represent *declaration by association*, and will be called *association types*. The informal meaning of the type expression `like` *anchor* is that it represents the type of *anchor*, where *anchor* is an attribute or the pseudo-variable `Current`. Since the type of an attribute may be changed during inheritance, association types are naturally understood as a way to use attributes as *bounded type variables*. The type variable is bounded because of the type redefinition rule, which ensures that an attribute can be redeclared only to a type that conforms to the original type. The expression `like Current` is a special case of this construction, which automatically takes on a new value during inheritance.

3. ATTRIBUTE TYPE REDEFINITION

The type redefinition rule allows attributes to be redeclared during inheritance, yet an inherited method may assign a value of the original type to the attribute, resulting in a dynamic violation of the type system.

The example below illustrates the problem of assignment to an attribute that is redeclared. The class `P1` defines an attribute `a` of type `Base`, and a method `setup` that assigns a new instance of `Base` to the attribute `a`. The subclass `C1` redeclares the attribute to have the type `Extra`, which conforms to `Base` in accordance with the type dedefinition rule. It also defines a `problem` method which calls the inherited `setup` method to initialise `a`, and then accesses the `extra` feature of `a`.

```
class P1 feature
  a : Base;

  setup is
    local
      x : Base;
    do
      x.Create;  -step 2: create an instance of
                    Base
      a := x;  -step 3: assign the new instance to a
    end;
end

class C1 inherit P1 redefine a feature
  a : Extra;

  problem : Integer is
    do
      setup;  -step 1: call setup (see above)
      Result := a.extra(2);  -step 4: a is an
  instance of Base
    end;
end
```

The comments describe the steps that occur when the `problem` message is sent to an instance of `C1`; a dynamic violation of the type system is produced in step 4.

The source of the problem is an interaction between assignment and attribute redeclaration. An assignment that is type-correct given the original declaration is no longer valid in the context of the redeclaration. In the face of this very fundamental problem, most programming languages have been forced to prohibit redeclaration of attributes.

Eiffel, however, has a way of getting round the problem of attribute redeclaration to some extent with declaration by association. If the declaration `x: Base` were changed to `x: like a` then the problem would not arise, because `setup` would be sensitive to possible redeclaration of `a`. Some problems with declaration by association are discussed in Section 7; the solution we propose obviates the need for redeclaration of attributes, replacing it by rebinding of an explicit type variable.

4. METHOD ARGUMENT TYPE REDEFINITION

The restriction of a formal method argument during inheritance, as permitted by the type redefinition rule, introduces the potential for dynamic type-errors within statically type-correct Eiffel programs. The problem arises when a subclass instance with a redefined argument is used where a superclass value is expected. In this situation an argument may be passed to the method that is not compatible with the redefinition, and a dynamic type-error may occur. In order to avoid type-errors, it is necessary that the original type conform to the new type (the conformance is in the opposite direction from result types); this property of function types is known as *contravariance*.[1]

The following example illustrates a dynamic type-error resulting from method redeclaration in a statically type-correct Eiffel program. Class P2 is declared to have a method get which calls the base method of its argument. The subclass C2 redeclares the get method and redefines it to call the extra method instead:

```
class P2 feature
  get(arg : Base) : Integer is
    do Result := arg.base(1) end:
end

class C2 inherit P2 redefine get feature
  get(arg : Extra) : Integer is
    do Result := arg.extra(2) end;
end
```

The example looks reasonable enough, until one considers that C2 conforms to P2 because C2 inherits P2. This means that objects created by C2 may be assigned to a variable of type P2. Manipulating a variable of type P2 which refers to an instance of C2 may lead to errors, as illustrated below:

```
local
  a : Base;
  v : P2;
  b : C2;
do
  a.Create;
  b.Create;
  v := b;   -statically type-correct because C2 conforms to P2
  v.get(a);                -call C2.get which calls a.extra(2)
end
```

Since v has static type P2, v.get has type Base→Integer, and v.get(a) is statically type-correct. But v contains an instance of C2 and methods are selected dynamically, so v.get refers to the get method of C2. This method, when passed a as a parameter will access the extra method of a, which does not exist.

This problem does not arise when redeclaring attributes, because assignment, although contravariant, is not part of the external interface of a class.

This type-error can also be created from within a class without using type-compatibility of assignment. If a redeclared method is called from within another parent method (sending a value that the parent considers appropriate), then dynamic type-errors may occur because the subclass method expects more specific arguments than the parent passes. This case is similar to the problem of attribute redefinition, where message sending is analogous to assignment. In this case, Eiffel is violating basic type-constraints on inheritance.[4]

The following example illustrates how the redeclaration may cause an error within a class. P2 is augmented by a problem method that creates an instance of Base and then uses the get method to extract its base feature.

```
class P2
  ...
  problem : Integer is
    local
      x : Base;
    do
      x.Create;
      Result := get(x);
    end;
end
```

A dynamic error occurs when an instance of class C2 is created and sent the problem message, because it will pass an instance of Base to the redefined get method which requires an instance of Extra.

Type-systems are conservative, in that they prevent the possibility of errors, but sometimes prohibit programs that do not in fact produce dynamic errors. This effect is an essential consequence of the locality of type-checking, and to prevent errors the only recourse is to invert the formal argument redefinition rule. The corrected type redefinition rule would read:

> A function result or a formal routine argument declared in a class may be redeclared with a new type in a descendant class, provided the new type conforms to the original one *if it is a result type, and the original one conforms to the new type if it is a formal argument type.*

This rule is used in other object-oriented languages,[7] but has the unfortunate effect of making argument type redefinition almost useless, since it is generally not very useful to redefine a method to accept a larger class of arguments.

5. METHODS DECLARED BY ASSOCIATION

Although changing the Eiffel type-redefinition rule would prevent run-time type-errors associated with contravariance, it would also impact the use of association types for formal argument declaration. This is because redeclaring an attribute indirectly causes the restriction of any formal argument associated with it, even though the type-expression representing the formal argument is not changed. In particular, if like Current is used, the method argument is effectively redeclared in any subclass, leading to the error outlined in Section 4.

We illustrate the problem using like Current, although the example could easily be changed to use an attribute as anchor. For this example the features of Base and Extra are combined with the classes

illustrating method redeclaration in Section 4. This combination allows the use of a `like Current` argument in the method `get`:

```
class P3 feature
  base(n : Integer) : Integer is
    do Result : + n * 2 end;
  get(arg : like Current) : Integer is
    do Result : = arg.base(1) end;
end

class C3 inherit P3 redefine get feature
  extra(n : Integer) : Integer is
    do result : = n * n end;
  get(arg : like Current) : Integer is
    do Result : = arg.extra(2) end;
end
```

The example behaves just like the one in Section 4, and it produces a similar dynamic type-error:

```
local
  a : P3;
  v : P3;
  b : C3;
do
  a.Create;
  b.Create;
  v : = b;        - valid because C3 conforms to P3
  v.get(a);       - call C3.get which calls
                    a.extra(2)
end
```

In Section 4, when faced with this problem, we concluded that the only solution was to prohibit redeclaration. If this approach were adopted here, it would mean prohibiting the use of declaration by association for formal argument types. This seems unreasonable; the special nature of association types allows a more acceptable solution.

We propose to eliminate the conformance relation that allowed the assignment `v : = b`, an essential step in producing the dynamic error. We propose that if a class has an argument declared by association, an inheriting class does not conform to that parent if the type of the anchor is redefined. In particular, the type of `Current` is always redefined, so formal arguments of type `like Current` prevent conformance.

The problem of internal consistency of a class, illustrated in the second example of Section 4, is handled by the reformulation of association types presented in Section 7. The changes amount to requiring actual argument types to be identical to the association type specifying the formal argument.

The solution given above has a subtle effect upon the conformance of `like Current`. The problem is that the conformance rule allows `like Current` to conform to the type of `Current`, while this is exactly the conformance relation we are proposing to eliminate.

To illustrate this problem, consider the effect of adding the following features to the class `P3`. These features will then be inherited by `C3`.

```
  class P3
  not_assoc(arg : P3) : Integer is
    local
      x : P3;
    do
      x.Create;
      Result : = arg.get(x);
  subvert(arg : like Current) : Integer is
    do
      Result : = not.assoc(arg);
    end;
end
```

If `c` has type `C3` then executing `c.subvert(c)` produces a dynamic type-error because `not.assoc` calls `c.get(x)` where `x` is an instance of `P3`.

The solution to this problem is simple: in a class `A` with a formal argument declared by association, `like Current` does not conform to `A`. Note that eliminating the conformance relation from descendants of such classes has an effect on their use as the bound for quantification over all its subclasses. This happens primarily in bounded generic classes, an Eiffel feature not described in *Object-Oriented Software Construction* (Ref. 6). However, it may be possible to define a type-safe interpretation for such classes as bounds.[2]

6. CONFORMANCE OF GENERIC INSTANCES

The conformance rule for generic classes is too permissive in assuming that two applications of a generic class conform if the actual arguments conform. This assumption does not always hold true, so that statically type-correct Eiffel programs using generic classes may produce dynamic type errors.

Before examining this problem, however, it is useful to clarify a smaller problem in the statement of the conformance rule for class types. Y conforms to X if:

> (2) X and Y are class types, X has no generic parameters, and Y lists X in its inheritance clause;
> (3) X and Y are class types, X is of the form $P[U_1, U_2, \ldots, U_n]$, and the inheritance clause of Y lists $P[V_1, V_2, \ldots, V_n]$ as parent, where every V_i conforms to the corresponding U_i.

Clause (2) prohibits X from having generic parameters.

This restriction seems unreasonable, because it seems that if Y lists X in its inherits clause then Y should conform to X, independent of the form of X. Inheritance ensures that Y will have all the features of the type X, even if it is a generic class type.

Clause (3) prevents $Y = P[V]$ from conforming to $X = P[U]$ when V conforms to U, even though this seems to be in the spirit of the rule. The problem is that the rule is stated in terms of the inherits clause of Y, not in terms of Y itself. One reason why this restriction seems arbitrary is that defining a trivial class that is equal to $P[V]$ allows the conformance to obtain:

`class PV inherit` $P[V]$ `end`

According to the clause (3), `PV` conforms to $P[U]$, although $P[V]$ does not conform to $P[U]$. The following two clauses are proposed as a more intuitive and faithful characterisation of conformance in Eiffel.* Y conforms to X if:

* Additional evidence that the rule is incorrect comes from the Eiffel implementation (version 2.1), which does not obey the clauses as stated above.

(2′) X and Y are class types and Y lists X in its inheritance clause;

(3′) X and Y are class types, X is of the form $P[U_1, U_2, \ldots, U_n]$, Y is of the form $P[V_1, V_2, \ldots, V_n]$, and every V_i conforms to the corresponding U_i.

Combined with the transitivity clause (5) of the original conformance rule, these clauses cover all the cases of the original, and allow $P[V]$ to conform to $P[U]$ if V conforms to U.

This reformulation of the generic conformance rule does not eliminate the fundamental problem, it merely clarifies it. It is still incorrect to assume that one application of a generic class conforms to another simply because the actual arguments conform.

Using the generic class `Cell` from the Eiffel library, it is easy to construct a program which is syntactically correct but produces a dynamic type error. The class `Cell` is defined as follows:

```
class Cell[T] feature
  info : T;

  change_info (x : T) is
    do info := x end;
end
```

The following code illustrates the problem with generic conformance:

```
local
  x : Base;
  a : Cell[Base];
  b : Cell[Extra];
do
  x.Create;
  b.Create;
  a := b;  -legal because Cell[Extra] conforms
           to Cell[Base]
  a.change_info(x);  - set b (and a) to contain x
  b.info.extra(4);      -x.extra does not exist
end
```

Two cells `a` and `b` are declared to contain instances of `Base` and `Extra` respectively. The cell `b` is created and the variable `a` is made to point at the `b` cell. This is type-corrected because of case (3′) of the conformance rule. Finally, an instance of `Base` is inserted into `a` (the same cell as `b`), and retrieved from `b`. At this point the result has static type `Extra` but is actually an instance of `Base` at run-time.

To correct the generic class conformance rule, we suggest distinguishing the various kinds of context in which a formal type parameter in a generic class may appear. If the type parameter is used to type attributes or method return values, but is not used as the formal argument type of a method, it is a *covariant* parameter. If it is used only to type the formal arguments of methods, and never as the type of an attribute or the result-type of a method, it is *contravariant*. If the parameter is used as both a method argument type and as either an attribute type or a method result-type, it is *bivariant*. In the class `Cell` above, the parameter `T` is bivariant because it is both the type of the attribute `info` and also the formal argument type of the method `change_info`.

A conformance rule for generic class types may be formulated which depends upon all parameters being either co- or contravariant. A single bivariant parameter prevents conformance among the different instances of the generic class. Y conforms to X if

(3″) X and Y are class types, X is of the form $P[U_1, U_2, \ldots, U_n]$, Y is of the form $P[V_1, V_2, \ldots, V_n]$, and for all i either the ith formal parameter of P is covariant and V_i conforms to U_i, or it is contravariant and U_i conforms to V_i, or it is bivariant and U_i equals V_i.

This check is reasonable because it only involves the interface of P, not its internal implementation. But as a result of this rule many generic class types will have no conformance relation at all among their instances. For example, `Cell[`U`]` will only conform to `Cell[`V`]` if $U = V$.

7. ASSOCIATION TYPES

Association types have been mentioned several times as a way to preserve expressive power when restrictions are proposed to make Eiffel type-safe. Unfortunately, declaration by association has problems of its own which call for a complete reformulation.

The conformance rule for association types prohibits some type-safe expressions. The first problem appears when using an instance, in calls to a method with a formal argument declared by association. According to the conformance rule, such a method can only be used on an actual parameter of exactly the same association type, because no other types conform *to* an association type. The problem is that the conformance rule for association types is designed to protect against errors that occur when attributes are redeclared, but types in the interface of an instance are fixed and cannot be changed. To illustrate, consider that the last line in this program fragment is not type-correct according to the Eiffel rules:

```
class C feature
  a : T;
  m (arg : like a) ...
end
local
  c : C;
  b : T;
do
  c.m(b);  - illegal because T does not conform to
             like a
```

This problem is corrected by viewing the type of an instance as having all its association types replaced by the types they stand for. This is merely a clarification of the typing rules, not a serious problem.

A related problem has to do with the relationship between an *anchor* and the type `like` *anchor* within a class definition. As it stands, the conformance rule prevents *anchor* from being assigned to a variable of type `like` *anchor*, or passed to a method where the formal argument has type `like` *anchor*. To illustrate, consider the following fragment:

```
class C feature
  x : T;
  y : like x;
  test is
    do y := x end;  - illegal because T does not
                      conform to like x
end
```

The assignment `y := x` is type-correct only if the type of `x`, i.e. `T`, conforms to the type of `t`, i.e. `like x`. But in general `T` does not conform to `like x` because `x` might be redeclared so that the conformance fails, as noted by Meyer.[6] However, assignment of the anchor `x` itself should be allowed because `like x` is always equal to 'the type of `x`'.*

To solve these problems, explicit *type attributes* are suggested as a replacement for declaration by association. An explicit type attribute is a type variable that may be redefined during inheritance to a type that conforms with its previous definition. The declaration `t = T` specifies a type attribute `t` and binds it to the type `T`. Using this idea, the declarations above may be rewritten as

```
t = T;
x : t
y : t
```

In this form, both assignments `x := y` and `y := x` are valid. Type attributes are closely related to generic classes, in that the example above is equivalent to the following one using genericity:

```
class GC[t ⩽ T] feature   – abstract over the type
                            attribute (with bound)
  x : t
  y : t ...
end

class C inherit GC[T] end  – bind the parameter
                             to the 'default' value
```

Meyer's comparison of inheritance (with declaration by association) and genericity should be re-examined in this light.[5]

The careful reader will observe that the special association type `like Current` cannot be explained by type attributes. `like Current` represents a truly novel aspect of Eiffel of both theoretical and practical interest.[2,3]

8. THE EXPORT MECHANISM

The independence of exportation and inheritance is subverted by the Eiffel conformance rules. Any feature that is exported in any ancestor of a class cannot be effectively hidden by omitting it from the export clause. These features can always be accessed by simply assigning the instance to a local variable whose type is the ancestor that exports the feature, and then accessing the feature using the local variable.

Access to 'hidden' features that are exported by an ancestor class is easily illustrated. If the class `Extra` defined in Section 2 were defined to export `extra` but not the inherited feature `base`, then for a variable `e: Extra` the expression `e.base` would be illegal. However, it is easy to access the value of `e.base` by the following sequence:

```
local
  v : Base;
do
  v := e;
  Result := v.base;
end
```

* Additional evidence that the type rules are incorrect is provided by the Eiffel compiler, which allows the assignment.

This trick can be used whenever access is needed to a feature that is exported in an ancestor but not in its child.

The example above demonstrates that in some sense exportation is cumulative: all the exported features of ancestors are available regardless of what their descendants try to specify in their export clauses. In view of this, we recommended that Eiffel drop the pretence of allowing inheritors to prevent access to features exported by an inherited class. With this change, classes would automatically inherit the export lists of their parents, and would only need to list those locally defined features which are to be exported.

9. CONCLUSION

Some problems in the Eiffel type system have been described and solutions proposed. Our recommendations for the Eiffel language are summarised below.

(1) Replace declaration by association (except the special type `like Current`) with explicit type attributes, which act like bounded generic parameters with default values.

(2) Eliminate attribute type redefinition and invert the method argument redefinition rule.

(3) Use the variance of generic parameters, type attributes and `like Current` to determine what conformance relations arise from inheritance.

(4) Determine the type of a class instance by replacing all generic parameters by the type to which they are bound.

(5) Make export clauses cumulative if the subclass conforms to its parent.

Perhaps the most significant aspect of these recommendations is the elimination of unsafe conformance relations between subclasses and their parents. On a practical level, it is uncertain to what degree existing Eiffel programs would be affected by this change. There are essentially three possibilities. (1) A program does not use unsafe conformance relations. With minor syntactic changes, it would remain valid after the proposed changes. (2) A program uses the conformance relations but does not produce execution errors. The program would not be valid after the corrections to Eiffel. Such programs could probably be made type-correct by simple changes, for example by introducing additional types. (3) A program is type-correct but produces dynamic errors. The corrections to the language should be useful in detecting the cause of the run-time errors.

From a theoretical point of view, it is significant that a subclass does not always conform to its parent. In Eiffel the conformance hierarchy is based on inheritance, but inheritance may create subclasses that do not conform, so the conformance hierarchy is a subset of the interface hierarchy. This means that the inheritance and conformance hierarchies are distinct. This conclusion is especially significant because it is a consequence of contravariance interacting with generic parameters or `like Current`, rather than deletion of messages.[8]

Acknowledgement

The author would like to thank Walter Olthoff, Walt Hill, Peter Canning and Alan Snyder for comments on this paper.

REFERENCES

1. L. Cardelli, A semantics of multiple inheritance. In *Semantics of Data Types*, Lecture Notes in Computer Science 173, pp. 51–68. Springer-Verlag, Heidelberg (1984).
2. P. Canning, W. Cook, W. Hill, J. Mitchell and W. Olthoff, F-bounded polymorphism for object-oriented programming. Proc. of Conf. on Functional Programming and Computer Architecture. (1989).
3. P. Canning, W. Cook, W. Hill and W. Olthoff. Interfaces for strongly typed object-oriented programming. Proc. of Conf. on Object-oriented Programming: Systems, Languages and Applications. (1989).
4. W. Cook, A Denotational Semantics of Inheritance. PhD Thesis, Brown University (1989).
5. B. Meyer, Genericity versus inheritance. Proc. of Conf. on Object-oriented Programming: Systems, Languages and Applications. pp. 391–405 (1986).
6. B. Meyer, *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ (1988).
7. Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian and Carrie Wilpolt, An introduction to Trellis/Owl. Proc. of Conf. on Object-oriented Programming: Systems, Languages and Applications. pp. 9–16 (1986).
8. Alan Snyder, Encapsulation and inheritance in object-oriented programming languages. Proc. of Conf. on Object-oriented Programming: Systems, Languages and Applications. pp. 38–45 (1986).