ISSN 0105-8517

Substitution Polymorphism for Object-Oriented Programming

Jens Palsberg
Michael I. Schwartzbach

DAIMI PB – 305
February 1990

Substitution Polymorphism for Object-Oriented Programming

Jens Palsberg [*] Michael I. Schwartzbach [†]

Computer Science Department, Aarhus University
Ny Munkegade, DK–8000 Aarhus C, Denmark

Abstract

We introduce substitution polymorphism as a new basis for typed object-oriented languages. While avoiding subtypes and polymorphic types, this mechanism enables optimal static type-checking of generic classes and polymorphic functions. The novel idea is to view a class as having a family of implicit subclasses, each of which is obtained through a substitution of types. This allows instantiation of a generic class to be merely subclassing and resolves the problems in the EIFFEL type system reported by Cook. All subclasses, including the implicit ones, can reuse the code implementing their superclass.

1 Introduction

This paper proposes a new and surprisingly simple basis for typed object-oriented languages, called *substitution polymorphism.*

With this mechanism, together with inheritance, we obtain a type system without subtyping, type variables or second-order entities. Even so, it enables static type-checking while generalizing parameterized classes, allowing a functional programming style with polymorphic functions, and solving the problems in the the EIFFEL type system that were reported by Cook [5]. It also separates the issues of polymorphism and heterogeneous data structures.

Inheritance and substitution polymorphism complement each other as subclassing mechanisms, see figure 1. Inheritance allows the construction

[*]Internet address: `palsberg@daimi.dk`
[†]Internet address: `mis@daimi.dk`

of subclasses by adding variables and procedures, and replacing procedure bodies. Substitution polymorphism allows the selection of subclasses by replacing types of variables and parameters. Those subclasses obtained by inheritance we call *explicit*, and those obtained by substitution we call *implicit*.

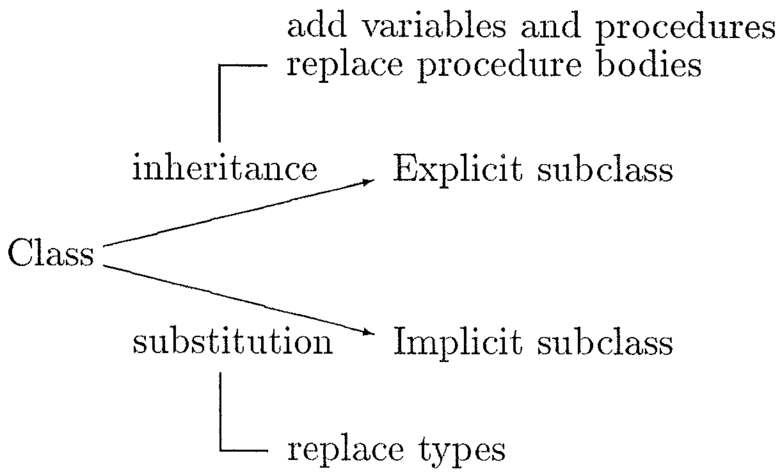

Figure 1: Inheritance and substitution.

In substitution polymorphism, a class yields a single type, identified by its name, and an object has only the type denoted by its class.

The construction of a subclass, using either inheritance or substitution, enjoys the following three properties.

- **Stability.** Equal types remain equal.
- **Monotonicity.** The type (i.e., class name) of a variable or parameter will only be substituted by subclass names.
- **Reusability.** The code compiled from the superclass can be reused.

In the following section we outline a core language without polymorphism. In section 3 we discuss polymorphism in object-oriented programming in general and clarify the differences between substitution, parametric, and inclusion polymorphism. In section 4 we develop a notation for the implicit subclasses. In section 5 we show how to program with substitution polymorphism and explain its relation to inheritance, genericity, declaration by association, and virtual classes, and note that it solves the problems in the EIFFEL type system that were reported by Cook [5]. In section 6 we show that polymorphic procedures declared outside classes can be provided as a shorthand, allowing a functional programming style. In section 7 we discuss the separation of polymorphism and heterogeneous data structures, and show that assignments involving different types are only needed when programming the latter. In section 8 we give the complete type-checking rules and state a soundness and optimality result.

Throughout, we use examples which are reformulations of some taken from Meyer's paper on genericity versus inheritance [16], Sandberg's paper on parameterized and descriptive classes [21], and Cook's paper on problems in the EIFFEL type system [5].

2 The Core Language

To avoid purely syntactic issues, we use a core language with PASCAL-like syntax and informal semantics, inspired by SIMULA [9], C++ [26], and EIFFEL [17]. The major aspects, except polymorphism, are as follows.

Objects group together variables and procedures, and are instances of classes. *Classes* are explicitly organized in a subclass hierarchy where a subclass inherits its superclass and may add variables and procedures, and reimplement procedure bodies. The built-in classes are object (the root of the subclass hierarchy), boolean, integer, and array. The last three cannot be inherited. Variables and parameters must be declared together with a *type*, which is a class name. In assignments and parameter passings, types must be equal. Recursive occurrences of the superclass name are in a subclass implicitly substituted by the name of the subclass. In procedures returning a result (functions), the variable Result is an implicitly declared local variable of the procedure's result type; its final value becomes the result of a call. When a variable is declared, an instance of the variable's class is notionally created. In an implementation, heap space is only allocated when dynamically needed, i.e., the first time the instance receives a message. This technique ensures that variables are never nil; we also avoid a new (create) statement. Finally, *class synonyms* are introduced using the transparent **let** name = class-name.

Let us now examine three approaches to introducing polymorphism into this core language.

3 Polymorphism

Object-oriented programming strives to obtain reusable software components. A key technique for doing this is to use languages which allow polymorphism. SMALLTALK's objects and methods, for example, are polymorphic because an object of a subclass can appear wherever an object of one of its superclasses is required. Flexibility is achieved by deferring to run-time all checks of whether objects understand the messages sent to them (instance variables are not declared with a type) [11].

Explicit type information makes programs easier to understand and allows a compiler to catch type errors and generate optimized code. Our core language, for example, can be statically type-checked but does not allow polymorphism because types must be equal in assignments and parameter passings.

In typed languages, *parametric* and *inclusion* polymorphism are the two major approaches to "universal" polymorphism, i.e., where procedures may be applied to arguments of an infinite number of types [4]. These techniques have been the basis of most attempts to introduce type polymorphism into object-oriented languages—hardly surprising in view of their acknowledged success in other applications.

The first approach is parametric polymorphism where procedures and classes have type parameters which may be instantiated to specific argument types [18]. Together with parameterized classes, Sandberg introduces descriptive classes as an *alternative* to subclassing [21]. Descriptive classes are used to avoid passing procedure parameters. Ohori and Buneman combine parameterized classes and inheritance with static type inference, though disallowing reimplementation of inherited procedures [19]. Language designs with both parameterized classes and inheritance include EIFFEL [17], TRELLIS/OWL [22], and DEMETER [14]. In all cases, a parameterized class yields a polymorphic type, i.e., a second-order entity which may be instantiated to specific types.

The other approach is inclusion polymorphism where objects may have more than one type [3]. In object-oriented languages, type systems are typically chosen such that a type may be a subtype of (conform to) other types. Objects are then viewed as having both the declared type and its supertypes. In our core language we use classes as types. It has been argued that classes and types should be distinct notions since classes describe implementation and types describe specification [25, 2].

In particular, it has been shown that if we use classes as types then a subclass need *not* yield a subtype [6]. Suggestions for fixing this by giving additional or alternative conformance rules have been given by Horn [12] (the notion of enhancement) and Cook [5] (in connection with the EIFFEL type system), but unfortunately the rules seem to be too complicated to fall into the mainstream of type systems. A type system of explicit object interfaces, independent of classes, has been proposed by Canning, Cook, Hill, and Olthoff [2]. Object interfaces can be parameterized and conform to others.

A significant drawback of parametric polymorphism is that polymorphic type instantiation does *not* correspond to subclassing. This makes it awkward to, for example, declare a class ring, then specializing it to a class matrix, and finally specializing matrix to a class booleanmatrix. A significant drawback of inclusion polymorphism based on subtyping is that the recursive types normally used in object-oriented programming have very few useful subtypes, due to the problematic contravariance of function types; for several demonstrations of how this hampers programming we refer to [8]. For example, one can *not* always obtain a subtype of a recursive record type by adding a field.

Recently, Walter Hill's group at HP Labs introduced the notion of F-bounded polymorphism [8, 2, 6]. It is a generalization of bounded quantification in which the bound type variable may occur within the bound. It characterizes types with similar recursive structure—types that need *not* be in subtype relation at all. The approach provides an improved typing of polymorphic procedures, compared to traditional bounded quantification [4], but still suffers the drawback of parametric polymorphism that polymorphic type instantiation does not correspond to subclassing.

Substitution polymorphism provides a new approach that has none of these drawbacks. With this technique, a class yields a single type, and an object has only the type denoted by its class. But every class also yields a family of implicit subclasses, all of which are obtained by substitutions. It is now possible to emulate the instantiation of a parametrized subclass by the selection of an appropriate implicit subclass. This allows a gradual specialization as well as a *post-hoc* parametrization of classes, both of which are very supportive of real-life software developments.

A summary of the differences between parametric, inclusion, and substitution polymorphism is provided in figure 2.

	Parametric polymorphism	Inclusion polymorphism	Substitution polymorphism
An object has a	single type	type + supertypes	single type
A class yields a	polymorphic type	single type	single type + family of subclasses

Figure 2: Three approaches to polymorphism.

4 Substitution Polymorphism

An explicitly constructed subclass inherits its superclass and may add variables and procedures, and reimplement procedure bodies. The effect can be explained in terms of the behavior of an (imagined) interpreter [11, 7]. If a procedure `p` in an object `x` is called then the interpreter must find the appropriate code to execute. This is done by finding the declared type of `x` and, starting in the corresponding class, searching towards `object` for an implementation of `p`.

When constructing such a subclass, one may *not* alter the types of variables or parameters. Such modifications are, however, realized by the implicit subclasses. They encompass all possible versions of the original class where types (i.e., class names) have been substituted by subclass names in a way such that equal types remain equal. By the way, notice that also inheritance lets equal types remain equal because *all* occurrences of the superclass name are substituted by the subclass name.

The effect of an implicit subclass can also be explained in terms of the interpreter's behavior. The search for `p` will no longer start in the class corresponding to the *declared* type of `x` but rather in the class corresponding to its *substituted* type.

The names of the implicit subclasses are *not* known to the programmer. In general, every class will have infinitely many implicit subclasses. It may be useful to think of their names as, say, combinations of the original class name and serial numbers. To enable the programmer to make use of these implicit subclasses, we shall develop a convenient notation for selecting the desired ones. The details of this development are presented in [20].

Definition 1 A class name A is said to *occur* in a class C if

1) it appears as the type of a variable or parameter; or
2) it occurs in a class corresponding to one of these types.

Notice that the definition is recursive.[1]

Proposition 1 Assume that the class name A occurs in the class C, and that B is a subclass of A. Then there exist implicit subclasses of C in which all occurrences of A have been substituted by B. Among these, there is a ***least*** specialized one from which all the others can be obtained through further specializations.

Of course, many other substitutions may have been necessary to maintain equality of types in these implicit subclasses.

Definition 2 If B is a subclass of A then $\mathsf{C}[\mathsf{A} \leftarrow \mathsf{B}]$ denotes the least specialized implicit subclass of C in which all occurrences of A have been substituted by B. If A does not occur in C then it denotes C itself.

As we shall see, this notation is easy to use.

Proposition 2 There is an algorithm that, given A, B, and C, computes the substituted types in $\mathsf{C}[\mathsf{A} \leftarrow \mathsf{B}]$.

Thus, a compiler can select the appropriate implementation when translating procedure calls.

Proposition 3 All implicit subclasses of C can be expressed as $\mathsf{C}[\mathsf{A}_1 \leftarrow \mathsf{B}_1]\ldots[\mathsf{A}_n \leftarrow \mathsf{B}_n]$, for some A_i and B_i.

Hence, we can base our language on the above notation without limiting ourselves.

Proposition 4 For any two notations of the above form, there is an algorithm to decide if the two implicit subclasses they denote are in a subclass relation to each other.

This ensures that we need never concern ourselves with any concrete names of the implicit subclasses.

```
class C1 inherits object
  var x,y: object
end
let C2 = C1[object ← integer]
```

Figure 3: Basic substitution polymorphism.

```
class D1 inherits object
  var c: C1
  proc p(arg: object)
    begin c.x:=arg end
end
let D2 = D1[C1 ← C2]
```

Figure 4: Derived substitutions.

For a simple example, see figure 3. C2 is a name for the implicit subclass of C1 where x and y have type integer. Clearly, one would obtain an implicit subclass with any class name in place of integer.

As mentioned, seemingly simple substitutions may lead to other *derived* substitutions that are required to maintain equality of types. For example, see figure 4. The declaration of D2 is legal because C2 is a subclass of C1. If types are to remain equal in the assignment then object must be substituted by integer. The notation D1[object ← integer] denotes the *same* implicit subclass of D1 as does D1[C1 ← C2] and, accordingly, they yield the same type.

In the following sections we show how to use substitution polymorphism when programming.

5 Programming with Substitutions

In this section we show how substitutions help to solve a number of standard problems from the literature.

Consider for example the stack classes in figure 5. In stack, the element type is object, and likewise the formal parameter of push and the result of top are of type object. The assignments in stack are therefore legal. Class booleanstack and integerstack are two implicit subclasses of

[1]The propositions in this section can all be formalized in terms of the *occurrence tree* of a class, defined in analogous manner.

```
class stack inherits object
   var space: array of object
   var index: integer
   proc empty returns boolean
      begin Result:=(index=0) end
   proc push(x: object)
      begin index:=index+1; space[index]:=x end
   proc top returns object
      begin Result:=space[index] end
   proc pop
      begin index:=index-1 end
   proc initialize
      begin index:=0 end
end
let booleanstack = stack[object ← boolean]
let integerstack = stack[object ← integer]
```

Figure 5: Stack classes.

stack. For example, booleanstack is the class obtained from stack by substituting *all* occurrences of object by boolean, leaving all assignments legal. Thus, stack acts like a parameterized class but is just a class, not a second-order entity. This enables *gradual* instantiations of "parameterized classes", as demonstrated in the following examples.

Consider next the ring classes in figure 6. Again, ring acts like a parameterized class, but it is more complicated than stack because it yields a recursive class (ring appears in the definition), and because the definitions of the procedures are deferred. The class booleanring is then defined as a subclass of one of class ring's implicit subclasses. This illustrates how substitution polymorphism coexists with and complements inheritance. In the implicit subclass of ring all occurrences of object are substituted by boolean, and in class booleanring the inherited procedures are implemented appropriately. Note that the implicit substitution of ring by booleanring means that we do not need the association type like Current as found in EIFFEL [16, 17].

Consider finally the matrix classes in figure 7. Again, the class matrix is defined as a subclass of one of class ring's implicit subclasses, whose procedures it implements appropriately. Note that we, as opposed to

```
class ring inherits object
  var value: object
  proc plus(other: ring)
  proc times(other: ring)
  proc zero
  proc unity
end
class booleanring inherits ring[object ← boolean]
  proc plus
    begin value:=(value or other.value) end
  proc times
    begin value:=(value and other.value) end
  proc zero
    begin value:=false end
  proc unity
    begin value:=true end
end
```

Figure 6: Ring classes.

EIFFEL, do not need a dummy variable of type ring serving as an *anchor* for some association types [16, 17]. Class booleanmatrix is identified as the implicit subclass of matrix with occurrences of ring substituted by booleanring, and consistently all occurrences of object substituted by boolean. Class matrixmatrix is analogous.

At this point, it may be worthwhile to review what the implicit subclass of ring denoted by ring[object ← array of array of ring] looks like. For purposes of illustration we will assume that the name of this subclass is known, and is in fact ring000127. It will then have a definition as found in figure 8.

The BETA language offers ***virtual classes*** as an alternative to generic types [15, 13]. Virtual class attributes may be substituted by descendants in subclasses, thus simulating substitution polymorphism. The explicit naming of virtual classes, however, allows inconsistent substitutions, e.g., in the same subclass some object's may be substituted by integer while others get substituted by boolean and still others do not get substituted at all. As seen in figure 7, a "high-level" substitution may imply other, derived substitutions. Using virtual classes, the programmer must de-

```
class matrix inherits ring[object ← array of array of
ring]
  proc plus
    var i,j: integer
    begin
      for i:=1 to arraysize do
        for j:=1 to arraysize do
          value[i,j].plus(other.value[i,j])
    end
  ...
end
let booleanmatrix = matrix[ring ← booleanring]
let matrixmatrix = matrix[ring ← matrix]
```

Figure 7: Matrix classes.

```
class ring000127 inherits object
  var value: array of array of ring
  proc plus(other: ring000127)
  proc times(other: ring000127)
  proc zero
  proc unity
end
```

Figure 8: A look behind the curtains.

termine *all* of these substitutions and specify them individually. Both virtual and parameterized classes require a prior knowledge of which components of a class may be eligible for later specialization. Substitution polymorphism allows for the *post hoc* parametrization, or virtualization, of a class.

Two aspects of substitution polymorphism are very compatible with the real-life process of software development. Firstly, every type occurring in a class can be viewed as a parameter for a *bounded parametric* class, and the programmer can through substitutions decrease these bounds dynamically. This provides a method for refining old generic classes to new generic ones which may be further specialized by subsequent subclassing. This is illustrated by the above development of rings and matrices. Secondly, the ability to view every type as a potential parameter should be very helpful. Not everything can be predicted in advance, and it is very awkward to go back and restructure an existing class hierarchy to introduce generic classes. The ability to perform arbitrary substitutions, rather than predicted instantiations, greatly increases the programmer's room for maneuvering. The simple rule of thumb to never use a more specialized class than is necessary will ensure the maximal possibilities for later, perhaps unforeseen, code reuse.

Substitution polymorphism solves the problems in the EIFFEL type system that were reported by Cook [5]. Using substitution, attributes cannot be redeclared in *isolation* in subclasses, there are no *asymmetries* as with declaration by association, and generic class instantiation has an equivalent formulation which yields a subclass. Our solution to the problem connected with contravariance of function types and assignment is presented in the section on heterogeneous data structures.

Substitution polymorphism not only allows classes to be polymorphic, it also provides polymorphic procedures which we consider next.

6 Polymorphic Procedures

Polymorphic procedures declared outside classes can be provided through substitution on-the-fly. This unifies to a large extent object-oriented and functional programming, although procedures cannot be returned as results. Our approach differs from that of Goguen and Meseguer [10] by dealing with imperative features, such as assignment, but not relational (logic) programming language features.

Consider, for example, the swap procedure in figure 9.

```
proc swap(inout x,y: object)
  var t: object
  begin t:=x; x:=y; y:=t end
```

Figure 9: Swap procedure.

When swap is called with two objects of the same type, the compiler will infer that it would have been possible to write the program in the following way:

1) Place the procedure in an auxiliary class with no other procedures or variables.

```
class order inherits object
  var value: object
  proc equal(other: order) returns boolean
  proc less(other: order) returns boolean
end
class integerorder inherits order[object ← integer]
  proc equal
    begin Result:=(value=other.value) end
  proc less
    begin Result:=(value<other.value) end
end
proc minimum(x,y: order) returns order
  begin if x.less(y) then Result:=x else Result:=y end
```

Figure 10: Order classes and a minimum procedure.

2) Identify an implicit subclass of the auxiliary class where object is substituted by the type of the actual parameters.

3) Create an object of the subclass.

4) Perform a normal call to the object's procedure.

This inference is algorithmically decidable. Note that, in general, each parameter suggests a substitution. The compiler checks that they do not conflict, combines them, and performs the combination. Thus, such polymorphic procedures can be called without sending a message to an object. Actual parameters can be instances of subclasses of the formal parameter types, but if two formal parameter types are equal then the corresponding two actual parameter types must equal as well. This parallels the developments in [23, 24].

Consider next the order classes and the minimum procedure in figure 10. Instances of order may be compared for equality and inequality, though in an asymmetrical way, as is usual in object-oriented programming. The minimum procedure is declared outside class order, is symmetrical, and takes two arguments of the same type provided the arguments are instances of a class which is a subclass of order. This gives an effect similar to bounded parametric polymorphism [4].

```
class list inherits object
  var empty: boolean
  var head: object
  var tail: list
end
proc cons(x: object; y: list) returns list
  begin
    Result.empty:=false;
    Result.head:=x;
    Result.tail:=y
  end
let orderlist = list[object ← order]
proc insert(x: order; y: orderlist) returns orderlist
  begin
    if y.empty or x.less(y.head)
    then Result:=cons(x,y)
    else Result:=cons(y.head,insert(x,y.tail))
  end
proc sort(x: orderlist) returns orderlist
  begin
    if x.empty
    then Result:=x
    else Result:=insert(x.head,sort(x.tail))
  end
let integerorderlist = orderlist[order ← integerorder]
```

Figure 11: List classes and a sort procedure.

As a final example, consider the list classes and the (insertion) sort procedure in figure 11. We have obtained the functional programming style by declaring procedures outside classes. The **sort** procedure takes an argument whose class is a subclass of **orderlist**. It gives back a list of the same type with the components of the argument sorted in ascending order.

These examples demonstrate the wide range of applications that are possible using substitution polymorphism while enabling static type-checking. We have shown that polymorphism can be obtained without resorting to assignments between unequal types. Programming heterogeneous data structures, however, demand a further extension of the core language, considered in the following section.

7 Heterogeneous Data Structures

It turned out that assignments between unequal types were never needed to achieve polymorphism or to construct generic classes. However, such assignments are clearly required to build heterogeneous data structures. This suggests that polymorphism and heterogenity are independent issues.

To obtain a general-purpose language, we now introduce "heterogeneous" variables, i.e., variables which may hold not only instances of the declared class but also those of its subclasses. They are declared as **var** name:< type. Such variables are needed for the programming of databases, for example, where instances of different classes are stored together. While allowing more programs, such variables disables compile-time type-checking. Run-time type-checking under similar circumstances were first used in SIMULA implementations, and later adopted in implementations of C++ and BETA.

```
class list inherits object
  var empty: boolean
  var head:< object
  var tail: list
end
```

Figure 12: A heterogeneous list class.

The list class in figure 12 is heterogeneous, since it contains a heterogeneous variable. All subclasses of list are again heterogeneous. When a class is heterogeneous then all variables of the corresponding type are automatically heterogeneous themselves. All polymorphic procedures declared outside classes can, however, be reused. Thus, the sort procedure does *not* have to be altered in any way.

Let us reexamine (a reformulation of) one of the EIFFEL programs that Cook provided in his paper on problems in the EIFFEL type system [5], see figure 13. Class parent specifies a procedure base and a procedure get which takes an argument of type parent and calls the base procedure of this argument. Class son is a subclass of parent and specifies in addition

```
class parent inherits object
  proc base
  proc get(arg: parent)
    begin arg.base end
end
class son inherits parent
  proc extra
  proc get
    begin arg.extra end
end
var p: parent
var s: son
begin p:=s; p.get(p) end
```

Figure 13: Cook's example.

a procedure extra. It also reimplements procedure get to call instead the extra procedure of its argument (which in class son is of type son).

Cook notes that in EIFFEL it is (erroneously) statically legal to declare a variable of type parent, assign a son object to it (because in EIFFEL son conforms to parent), and then use the parent variable as if it referred to a parent object, for example by calling the referred object's get procedure with an argument of type parent. This will lead to a run-time error because when the get procedure in the son object is executed, it will try to access the extra procedure of its argument which does not exist.

Cook observes that the problem in the type system stems from considering that son conforms to parent; the restriction of the argument type of procedure get in class son violates the contravariance of function types.

In our view, the parent variable should be declared as heterogeneous in order to allow the assignment of a son object to it. This declaration also signals a warning that run-time checks may be necessary. When calling the referred object's get procedure, the compiler will know that the object need not be of type parent, and thus insert a run-time type-check of the argument (which will fail in this case).

Run-time type-check may also be needed when assigning a heterogeneous expression, for example when retrieving information from a database. In the following section we give the complete type-check rules and state a soundness and optimality result.

8 Optimal Type-checking

The traditional purpose of type-checking in object-oriented languages is to ensure that all messages to objects will be understood [1]. In the homogeneous subset of our language this can be entirely determined at compile-time. We propose the following static checks.

- **Early checks.** We verify for all message passings `x.p(...)` that a procedure `p` is implemented in the class corresponding to the declared type of the object `x`, or a superclass.

- **Equality checks.** We further verify for all assignments and parameter passings that the two declared types (left-hand/right-hand, formal/actual) are equal.

Proposition 5 Early checks and equality checks are sound and optimal, i.e., they are necessary and sufficient to ensure that all messages will be understood.
Proof sketch. Clearly, these checks are necessary. In their absence, it is quite easy to construct counter-example programs where some messages will not be understood. If the checks are satisfied then the explicitly written code is correct. We need to ensure that this correctness is preserved in all subclasses.

We recall two properties of the constructions of both implicit and explicit subclasses.

- **Monotonicity.** The type (i.e., class name) of a variable or parameter will only be substituted by subclass names.

- **Stability.** Equal types remain equal.

This enables us to perform an inductive arguments that subclasses are also correct. Assume that the superclass satisfies the checks. Then monotonicity guarantees that the *early* checks are satisfied in the subclass, and stability guarantees that the *equality* checks are satisfied in the subclass. Hence, the subclass will be correct, too. □

This settles the issue of type-checking in the homogeneous sublanguage. Actually, most parts of a program need only use homogeneous variables [1]. If heterogeneous variables are introduced then compile-time checks are no longer sufficient. One solution to this predicament is to switch entirely to run-time checks of individual messages, in the style of SMALLTALK. It is, however, a vast improvement to direct the attention towards assignments, which allows the mixture of compile- and run-time checking that is used in SIMULA, C++, and BETA. It turns out that in many cases, run-time checks can be entirely dispensed with.

First of all, the usual early checks are performed. Only the equality checks need to be revised. For this analysis, we can identify assignments and parameter passings. We now have four cases, as both the left- and right-hand object can be homogeneous or heterogeneous. Let stat(x) be the statically declared class of an object x and dyn(x) its dynamic class. If x is homogeneous then stat(x) = dyn(x), whereas if x is heterogeneous then stat(x) $\geq$ dyn(x). Here, $>$ indicates the ordering between super-classes and subclasses. We consider the assignment L:=R.

1) **L and R are both homogeneous:** (The case we handled above.) At compile-time we verify that the relation stat(L) = stat(R) holds.
2) **L is heterogeneous, R is homogeneous:** At compile-time we verify that the relation stat(L) $\geq$ stat(R) holds.
3) **L is homogeneous, R is heterogeneous:** At compile-time we verify that the relation stat(L) $\leq$ stat(R) holds. At run-time we verify that the relation stat(L) = dyn(R) holds.
4) **L and R are both heterogeneous:** If, at compile-time, stat(L) $\geq$ stat(R) then no run-time checks are necessary. If, at compile-time, stat(L) $<$ stat(R) then we verify at run-time that the relation stat(L) $\geq$ dyn(R) holds.

9 Conclusion

We have presented a new approach to polymorphism in object-oriented languages. It has none of the drawbacks of parametric and inclusion polymorphism and offers many pragmatic advantages, such as static type-checking, gradual instantiation, and polymorphic procedures.

We recommend that typed object-oriented languages, such as EIFFEL, adopt substitution polymorphism in place of for example generic classes and declaration by association. This would simplify language design and avoid the problems reported by Cook.

Acknowledgements. The authors thank Peter Mosses, Ole Lehrmann Madsen, and Jørgen Lindskov Knudsen for helpful comments on a draft of the paper.

References

[1] Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Ninth Symposium on Principles of Programming Languages*, pages 133–141. ACM Press, January 1982.

[2] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proc. OOPSLA'89, Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM, 1989.

[3] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, pages 51–68. Springer-Verlag (*LNCS* 173), 1984.

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), December 1985.

[5] William Cook. A proposal for making Eiffel type-safe. In *Proc. ECOOP'89, European Conference on Object-Oriented Programming*, 1989.

[6] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *Seventeenth Symposium on Principles of Programming Languages.* ACM Press, January 1990.

[7] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. OOPSLA'89, Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM, 1989.

[8] William R. Cook, Walter L. Hill, and Peter S. Canning. F-bounded polymorphism for object-oriented programming. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, 1989.

[9] O. J. Dahl, B. Myhrhaug, and K. Nygaard. Simula 67 common base language. Technical report, Norwegian Computing Center, Oslo, Norway, 1968.

[10] Joseph A. Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[11] A. Goldberg and D. Robson. *Smalltalk-80—The Language and its Implementation.* Addison-Wesley, 1983.

[12] Chris Horn. Conformance, genericity, inheritance, and enhancement. In *Proc. ECOOP'87, European Conference on Object-Oriented Programming.* Springer-Verlag (*LNCS* 276), 1987.

[13] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. The BETA programming language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[14] Karl J. Lieberherr and Arthur J. Riel. Contributions to teaching object-oriented design and programming. In *Proc. OOPSLA'89, Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM, 1989.

[15] Ole L. Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proc. OOPSLA'89, Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM, 1989.

[16] Bertrand Meyer. Genericity versus inheritance. In *Proc. OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications.* Sigplan Notices, 21(11), November 1986.

[17] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.

[19] Atsushi Ohori and Peter Buneman. Static type inference for parametric classes. In *Proc. OOPSLA'89, Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM, 1989.

[20] Jens Palsberg and Michael I. Schwartzbach. Substitution polymorphism and its semantics. Computer Science Department, Aarhus University. In preparation.

[21] David Sandberg. An alternative to subclassing. In *Proc. OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications.* Sigplan Notices, 21(11), November 1986.

[22] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proc. OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications.* Sigplan Notices, 21(11), November 1986.

[23] Erik M. Schmidt and Michael I. Schwartzbach. An imperative type hierarchy with partial products. In *Proc. of Mathematical Foundations of Computer Science 1989.* Springer-Verlag (*LNCS* 379), 1989.

[24] Michael I. Schwartzbach. Static correctness of hierarchical procedures. In *Proc. International Colloquium on Automata, Languages, and Programming 1990.* Springer-Verlag (*LNCS*), 1990.

[25] A. Snyder. Inheritance and the development of encapsulated software components. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[26] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.