ISSN 0105-8517

Type Substitution for Object-Oriented Programming

Jens Palsberg
Michael I. Schwartzbach

DAIMI PB – 317
July 1990

COMPUTER SCIENCE DEPARTMENT
AARHUS UNIVERSITY
Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark

ISSN 0105-8517

PB – 317 **Palsberg & Schwartzbach: Type Substitution**

Type Substitution for Object-Oriented Programming

Jens Palsberg   Michael I. Schwartzbach

Computer Science Department, Aarhus University
Ny Munkegade, DK–8000 Aarhus C, Denmark
Internet addresses: `palsberg@daimi.dk` and `mis@daimi.dk`

Abstract

Genericity allows the substitution of types in a class. This is usually obtained through parameterized classes, although they are inflexible since any class can be inherited but is not in itself parameterized. We suggest a new genericity mechanism, *type substitution*, which is a subclassing concept that complements inheritance: any class is generic, can be "instantiated" gradually without planning, and has all of its generic instances as subclasses.

©ACM. A reformatted version of this report is to be presented at *OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, to be held 21–25 October 1990, in Ottawa, Canada. Citations should refer to the Proceedings.

1 Introduction

This paper proposes *type substitution* as a new genericity mechanism for statically typed object-oriented languages. With this concept we avoid type variables and second-order entities and obtain the natural complement of inheritance; both are subclassing mechanisms and they differ as follows.

- **Inheritance.** Construction of subclasses by adding variables and procedures, and by replacing procedure bodies.
- **Type substitution.** Construction of subclasses by substituting types.

Type substitution is supportive of the real-life process of software development. Every type occurring in a class can be specialized through substitutions. This allows old generic classes to be refined to new generic ones which may be further specialized by subsequent subclassing. Also, every type can be viewed as a potential parameter. Not everything can be predicted in advance, and it is awkward to go back and restructure an existing class hierarchy to introduce parameterized classes. The ability to perform arbitrary substitutions, rather than predicted instantiations, maximizes possibilities for later, perhaps unforeseen, code reuse.

In the following section we outline a core language that will be used in examples. In section 3 we discuss code reuse, polymorphism, inheritance, and genericity. In section 4 we introduce type substitution as a new subclassing mechanism to complement inheritance. In section 5 we show how to program with type substitution, and note that it solves some problems in the EIFFEL type system that were reported by Cook [7]. In section 6 we show that polymorphic procedures declared outside classes can be provided as a shorthand, allowing a symmetric programming style. In section 7 we demonstrate the usefulness of opaque definitions. In section 8 we discuss heterogeneous variables. Finally, in section 9 we present the type-checking rules.

Throughout, we use examples which are reformulations of some taken from Meyer's paper on genericity versus inheritance [20], Sandberg's paper on parameterized and descriptive classes [26], and Cook's paper on problems in the EIFFEL type system [7].

2 The Core Language

To avoid purely syntactic issues, we use a core language with PASCAL-like syntax and informal semantics, inspired by SIMULA [11], C++ [31], and EIFFEL [21]. The major aspects are as follows.

Objects group together variables and procedures, and are instances of *classes*. The built-in classes are object (the empty class), boolean, integer, and array. Variables and parameters must be declared together with a *type*, which is a class. In assignments and parameter passings, types must be equal. 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. Extra class names can be specified in two ways: through the transparent

let name = class

which yields a synonym, and through the opaque

let name $\approx$ class

which yields new type with the same implementation.

Let us now examine different approaches to introducing code reuse into this core language.

3 Code Reuse

Object-oriented programming strives to obtain reusable software components, e.g., procedures, objects, classes. The two major approaches to this are *polymorphism* and *prefixing*, see figure 1.

- **Polymorphism.** A component may have *more* than one type. Examples: ML-functions [22], generic ADA-packages [12], parameterized CLU-clusters [17].

- **Prefixing.** A component may be described as an *extension* of another component. Examples: Delegation [16], SIMULA-prefixing [11], SMALLTALK-inheritance [13].

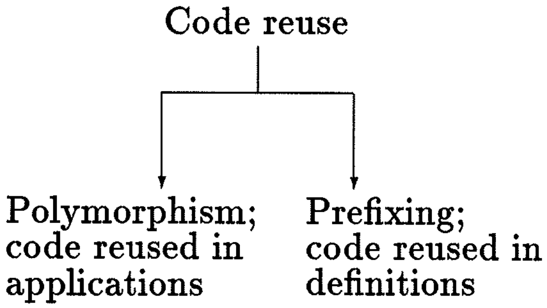

Figure 1: Two approaches to code reuse.

In typed languages, a component can be used only according to its type. A polymorphic component has several types; hence it can be used in several different ways. In this approach, code is reused in *applications.* If the behavior of a component is completely described by its type, then that type is unique and polymorphism is of course not possible. Alternatively, a description can be used as a prefix of other descriptions. In this approach, code is reused in *definitions.*

A class completely describes the behavior of its objects. Many object-oriented languages use classes as types, have inheritance as prefixing (subclassing) mechanism, and allow variables and assignments. Major examples are Simula, C++, and Eiffel. Inheritance gives a particular style of code reuse in definitions. It allows the construction of subclasses by adding variables and procedures, and by replacing procedure bodies. This has inspired a development of polymorphic languages seeking to obtain the same style of code reuse in applications. They are functional languages using object *interfaces* as types [30]. By allowing a type to be a subtype of (conform to) other types, an object can be viewed as having both the declared type and its supertypes. Hence, code applicable to objects of some type can also be applied to objects of a subtype.

Since Cardelli's seminal paper [3], the definition of the subtype relation has undergone a number of modifications to achieve a closer resemblance to inheritance. Bounded parametric polymorphism was introduced by Cardelli and Wegner [4] to avoid type-loss in applications, and recently F-bounded polymorphism [10, 2, 8] has been proposed to resolve a number of shortcomings involving recursive types. It should be noted, though, that these polymorphic languages cannot emulate inheritance of classes with variables because mutable types have no non-trivial sub-

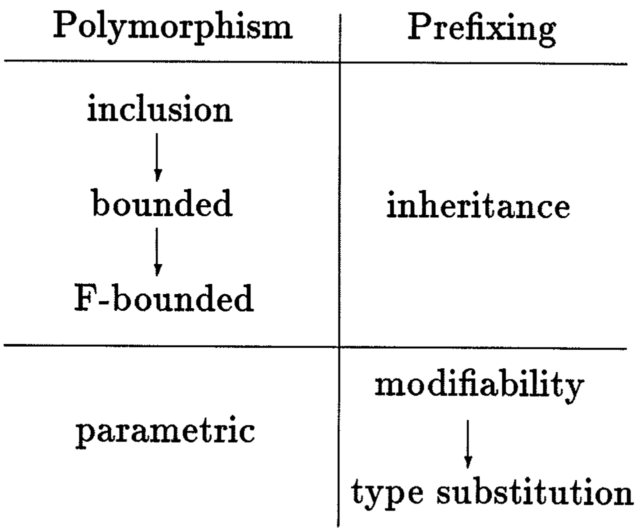

Figure 2: Polymorphism and prefixing.

types, as observed by Cardelli [5]; this is further discussed in the section on heterogeneous variables.

Some polymorphic languages allow *parameterized* types which give a different style of code reuse that cannot be expressed through inheritance. Parameterized types can be used to describe *generic* components, i.e., components whose type annotations can be substituted. This has inspired several attempts of providing a notion of generic class. The simplest approach is to use parameterized classes. A parameterized class is a second-order entity which is instantiated to specific classes when actual type parameters are supplied. Together with parameterized classes, Sandberg introduces descriptive classes as an *alternative* to subclassing [26]. 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 [23]. Language designs with both parameterized classes and inheritance include EIFFEL [21], TRELLIS/OWL [27], and DEMETER [15].

Instantiation of parameterized classes is less flexible than inheritance, since any class can be inherited but is not in itself parameterized. In other words, code reuse with parameterized classes requires planning; code reuse with inheritance does not. Another drawback of parameterized classes is that they cannot be gradually instantiated. 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.

Another approach to generic classes is the use of *modifiable* declarations, exemplified by the Simula [11] and Beta [18, 14] notion of virtual attributes. This technique allows types to be modified in a subclass, thus providing substitution of type annotations in a generic class as a subclassing mechanism. Unfortunately, individual conflicting modifications may yield type-incorrect subclasses; this leads to a fair amount of runtime type-checking [19], which is superfluous if the resulting class is in fact type-correct.

A summary of polymorphism and prefixing is provided in figure 2. In the following section, we introduce *type substitution* as a new approach to generic classes. It is a subclassing mechanism without the drawbacks of parameterized classes and modifiable declarations.

4 Type Substitution

Inheritance is not the only possible subclassing mechanism. This section discusses a more general notion of type-safe code reuse of class definitions, and suggests *type substitution* as a new subclassing mechanism to complement inheritance. The discussion is informal; formal definitions and proofs are given in [24].

```
class sequence
  var head: object
  var tail: sequence
end
```

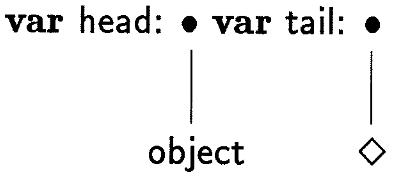

Figure 3: Sequences.

Let us first define the *universe* of all possible classes. A class can be thought of as *untyped* code in which type *annotations* are included whenever variables and parameters are declared. Since types are classes, this gives rise to a tree *denotation* of a class. The root is the untyped code of the class, and for each position where a type annotation is written,

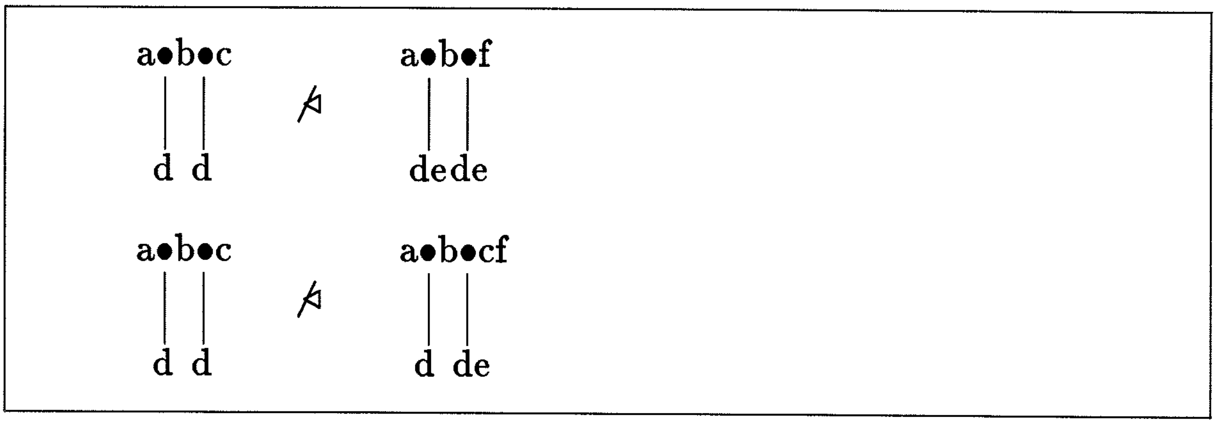

Figure 4: Monotonicity and stability fail.

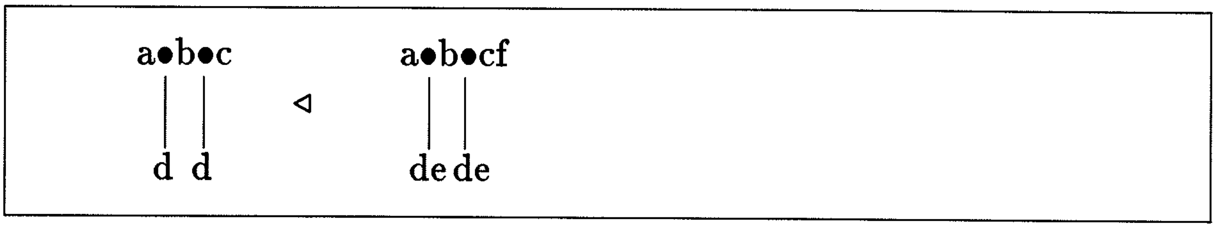

Figure 5: Monotonicity and stability.

there is a subtree with the tree denotation of the corresponding class. The only exception is that recursive occurrences of the class itself are not denoted explicitly, but by the symbol $\diamond$. Notice that a tree denotation may still be infinite if it contains other recursive types. As a simple example, a sequence class and its tree denotation is presented in figure 3. Note that the standard dynamic semantics of method lookup [9, 25, 6] can be based exclusively on these denotations. Thus we need only explain inheritance and type substitution by their effect on denotations; their dynamic semantics can then be inferred.

There is a relation $\lhd$ on tree denotations that indicates the possibilities for type-safe code reuse, i.e., if $T_1 \lhd T_2$ then T_1 may be reused in the definition of T_2. We need the following two requirements:

- Monotonicity: the code in T_1 must nodewise be a prefix of the code in T_2.

- Stability: if two types in T_1 are equal, then the corresponding two types in T_2 must be equal.

These requirements ensure that type-correctness of procedure calls and assignments in T_1 is preserved in T_2, as discussed in a later section on type-checking. Figure 4 shows how monotonicity and stability may

```
class list inherits sequence
  var empty: boolean
end
```

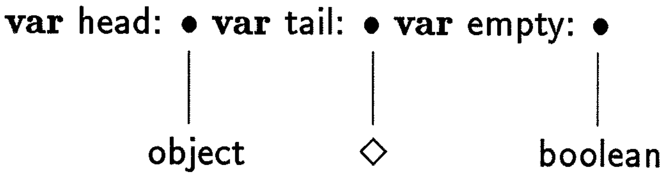

Figure 6: Lists.

```
sequence[object ← integer]
```

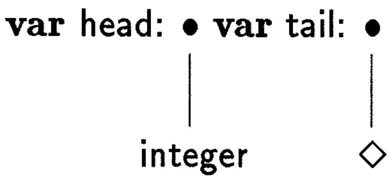

Figure 7: Integer sequences.

fail, whereas figure 5 illustrates a situation where both properties hold. The relation $\lhd$ is a decidable, partial order. We generalize the usual terminology by calling T_2 a subclass of T_1. When opaque definitions are considered, then $\lhd$ is only a preorder, i.e., a class and its opaque versions are different but mutually $\lhd$-related.

We observe that if T_2 inherits T_1, then it is indeed the case that $T_1 \lhd T_2$. Monotonicity holds since the root of T_1 is a prefix of the root of T_2. Stability holds since no type in T_1 is changed.

A simple example of inheritance is presented in figure 6. The use of $\diamond$ ensures that the subclass has the same recursive structure as the superclass it inherits. If instead the denotation was completely expanded, then only the first element of a list instance would have an empty component.

The key observation is that $\lhd$ contains possibilities for substituting type annotations that are not realized by inheritance. This leads us to introduce the following new mechanism. If C, A_i, and B_i are classes, then

$$\mathsf{C}[\mathsf{A}_1, \ldots, \mathsf{A}_n \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_n]$$

is a *type substitution* which specifies a class D such that $\mathsf{C} \lhd \mathsf{D}$ and all occurrences of A_i are substituted by B_i. As a simple example, consider the type substitution and its resulting denotation in figure 7.

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

Figure 8: *Not* textual substitution.

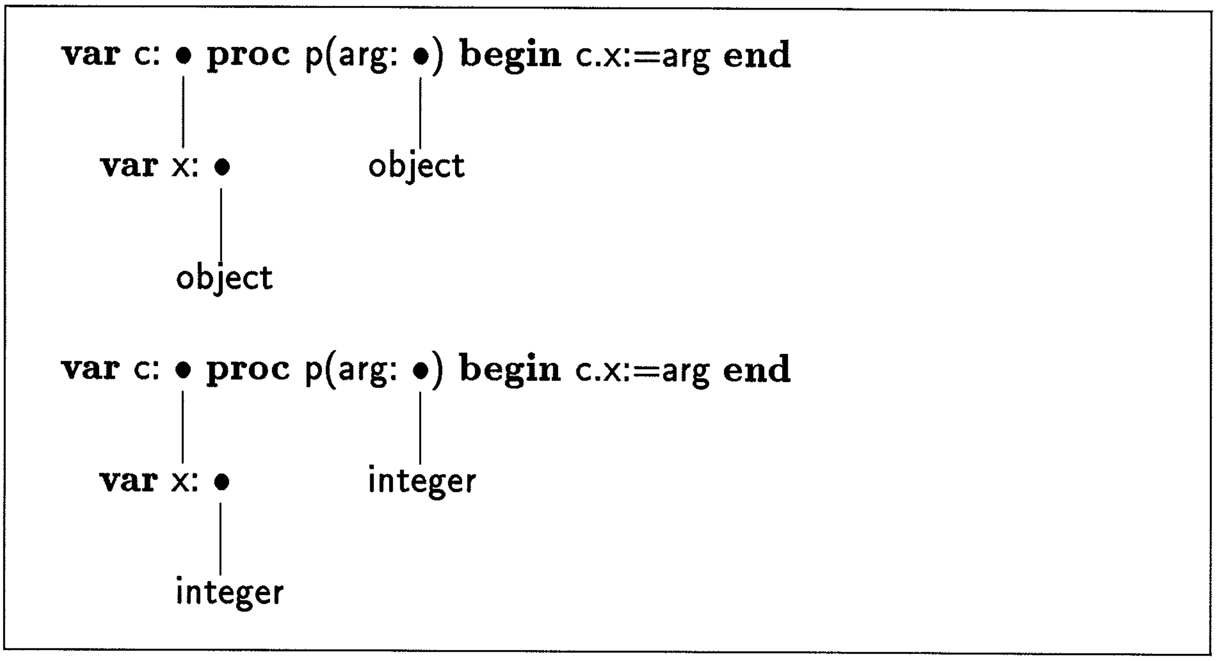

Figure 9: Denotations of D1 and D2.

Clearly, not all specifications can be realized. For example, we could not specify that object should be substituted by both boolean and integer. We say that a specification as the above is *consistent* when a tree with A_i-subtrees is $\lhd$-smaller than the same tree with B_i-subtrees instead. Every consistent specification can be realized by a unique most general class, i.e., one whose types are the least specialized; furthermore, this unique class can be computed from the specification, and is by definition $\lhd$-related to its superclass.

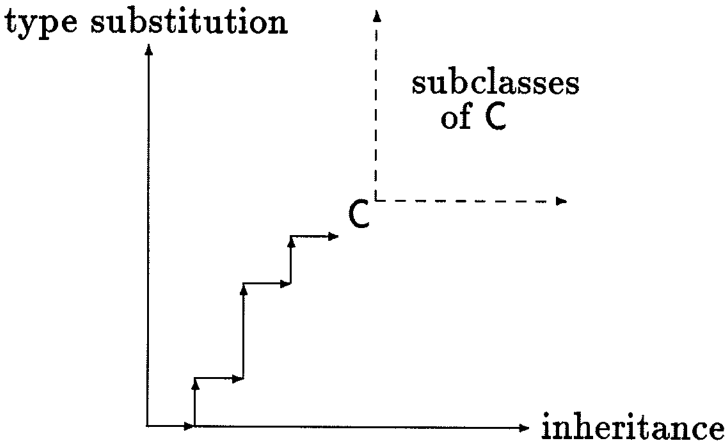

Figure 10: Two dimensions of subclassing.

In figure 7, type substitution appears indistinguishable from textual substitution. This, however, is only because the example is simple. In general, textual substitution will not yield a type-correct class, as illustrated by figure 8. If we try to obtain D2 as D1 with C1 textually substituted by C2, then the assignment in procedure p involves different types: an integer and an object. Using type substitution, D2 is the smallest type-correct subclass of D1 with C1 substituted by C2. This means that also object is substituted by integer. In fact, an equivalent specification of D2 is D1[object ← integer]. The relevant denotations are shown in figure 9.

We now have two mechanisms which exploits the $\lhd$-relation: inheritance and type substitution. Obviously, we must consider if there are more possibilities for code reuse left. It turns out that the answer is *no*: every subclass of a class C can be obtained through a finite number of inheritance and type substitution steps. Furthermore, inheritance and type substitution can be shown to form an orthogonal basis for $\lhd$, as indicated in figure 10. Thus, genericity and inheritance are reconciled as independent, complementary components of a unified concept [20]. Also, every subclass can in fact be obtained by one type substitution followed by one application of inheritance. The converse is false because the extra code may exploit the larger types.

Type substitution solves some problems in the EIFFEL type system that were reported by Cook [7], since attributes cannot be redeclared in *isolation* in subclasses, there are no *asymmetries* as with declaration by association, and parameterized class instantiation can be expressed as subclassing.

The following section shows how to program with type substitution.

5 Programming Examples

Consider the stack classes in figure 11. In stack, the element type is object, and likewise the formal parameter of push and the result of top are of type object. The classes booleanstack and integerstack are type substitutions of stack. For example, booleanstack is the class obtained from stack by substituting *all* occurrences of object by boolean, leaving all assignments legal.

```
class stack
  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 11: Stack classes.

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 recursive ring classes in figure 12. The class booleanring inherits a type substitution of class ring; thus, booleanring is a subclass of ring.

```
class ring
  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 12: Ring classes.

```
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 13: Matrix classes.

This illustrates how type substitution and inheritance complement each other: first object is substituted by boolean; then the inherited procedures are implemented appropriately.

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

Figure 14: Recursive structure is preserved.

Since the recursive structure of a class is preserved during type substitutions, we do not need the association type like Current as found in EIFFEL [20, 21].

This can be further illustrated by the matrix classes in figure 13. Again, the class matrix is obtained through a type substitution followed by an application of inheritance. Note that we, as opposed to EIFFEL, do not need a dummy variable of type ring serving as an *anchor* for some association types [20, 21]. In class booleanmatrix occurrences of ring are substituted by booleanring, and consequently occurrences of object are substituted by boolean. Class matrixmatrix is obtained analogously. That the recursive structure of a class is preserved in its subclasses can be seen by focusing on the class ring[object ← array of array of ring], which has the same denotation as the class in figure 14; the occurrence of ring in the type of value is not subsumed.

6 Polymorphic Procedures

Polymorphic procedures declared outside classes can be provided through type substitution. This allows symmetric operations and a more functional programming style.

Consider, for example, the swap procedure in figure 15. 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.

2) Identify a subclass where object is substituted by the type of the actual parameters.

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

Figure 15: Swap procedure.

```
class order
  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 16: Order classes and a minimum procedure.

3) Perform a normal call to the procedure in a notional object of the subclass.

Note that the formal and actual parameters specify the substitution. A call is, of course, only legal when this specification is consistent. 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 be equal as well. This parallels the developments in [28, 29].

Consider next the order classes and the minimum procedure in figure 16. 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
  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 17: List classes and a sort procedure.

As a final example, consider the list classes and the (insertion) sort procedure in figure 17. 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. Notice the polymorphic calls of cons, and that sort can be called with an integerorderlist.

7 Opaque Definitions

```
let arg ≈ object
let res ≈ object
class map
  var a: arg
  var r: res
  var next: map
  proc update(x: arg; y: res)
    begin ··· end
  proc inspect(x: arg) returns res
    begin ··· end
end
let phonebook = map[arg,res ← text,integer]
```

Figure 18: Use of opaque definitions.

The consistency condition on type substitutions seems at first to impose unwanted restrictions. We may have that two occurrences of e.g. object in a class are intended to play entirely different roles. However, in a type substitution they must be substituted by the same type to uphold consistency. Such problems can be avoided by judicious application of opaque definitions, as illustrated in figure 18.

In the class map we clearly want to allow arbitrary argument and result types. This suggests that they should both have type object; but we also want the types of arguments and results to be completely independent. By defining the types of arg and res to be opaque versions of object, we can achieve both aspirations simultaneously. The class phonebook can now be obtained through a type substitution of text for arg and integer for res.

8 Heterogeneous Variables

Assignments between unequal types were not needed to construct generic classes. Actually, most parts of a program do not need such assignments [1]. However, they are clearly required to build heterogeneous data structures. This suggests that genericity and heterogeneity are independent issues.

To obtain a comprehensive 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 disable compile-time type-checking. Run-time type-checking under similar circumstances were first used in SIMULA implementations, and later adopted in the implementation of BETA.

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

Figure 19: A heterogeneous list class.

```
class parent
  proc base
  proc get(arg: parent)
    begin arg.base end
end
class son inherits parent
  proc extra
  proc get(arg: son)
    begin arg.extra end
end
var p,q: parent
var s: son
begin
  p:=s;
  p.get(q)     (* run-time error *)
end
```

Figure 20: Cook's example.

The list class in figure 19 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.

Let us reexamine (a reformulation of) one of the EIFFEL programs that Cook provided in his paper on problems in the EIFFEL type system [7], see figure 20. 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 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 claims 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. But, since EIFFEL uses variables and assignments, an analysis based on subtyping is not appropriate, as noted in section 3. In our analysis, the parent variable p 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).

To further illustrate the problem with variables and subtyping, see figure 21. An execution of this program will lead to a run-time error because b.x is in fact of type object. In Cardelli's analysis, big is not a subclass of small since they both contain variables; hence, the call of switch with the actual parameter b is illegal. An analysis analogous to the ones of Cook [7] would erroneously deem the program legal since variables are ignored and only the usual subtyping rules are considered. In our analysis, the program is illegal because consistency fails in the call of switch: b and s have different types. Note that we will allow a

```
class small
  var x: object
end
class big
  var x: integer
end
proc switch(p,q: small)
  begin p.x:=q.x end
var s: small
var b: big
var i: integer
begin
  switch(b,s);
  i:=b.x+1     (* run-time error *)
end
```

Figure 21: Variables cannot be ignored.

call of switch(b,b); Cardelli prohibits this even though it will not lead to run-time errors. SIMULA and BETA's assignable variables can only be heterogeneous; hence superfluous run-time type-checks will be inserted by their implementations in such situations.

In the following section we give the complete type-check rules which also considers heterogeneous expressions.

9 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. The rules to check whether source code is type-correct are

- Early checks: verify for all calls x.p(...) that a procedure p is implemented by the declared type of x.
- Equality checks: verify for all assignments and parameter passings that the two declared types are equal.

Note that in theory these checks should be performed on the denotations of the classes. However, because of monotonicity and stability of $\lhd$, the validity of such checks will be preserved when code is reused through inheritance and type substitution. More specifically, monotonicity preserves early checks and stability preserves equality checks. Hence, source code need only be checked once, as usual.

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 and BETA. It turns out that in many cases, run-time checks are not needed anyway.

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) $\triangleleft$ dyn(x). We consider the assignment L:=R.

1) **L and R are both homogeneous:** At compile-time we verify that stat(L) = stat(R).

2) **L is heterogeneous, R is homogeneous:** At compile-time we verify that stat(L) $\triangleleft$ stat(R).

3) **L is homogeneous, R is heterogeneous:** At compile-time we verify that stat(L) $\triangleright$ stat(R). At run-time we verify that stat(L) = dyn(R).

4) **L and R are both heterogeneous:** If, at compile-time, stat(L) $\triangleleft$ stat(R) then no run-time checks are necessary. If, at compile-time, stat(L) $\triangleright$ stat(R) and stat(L) $\neq$ stat(R) then we verify at run-time that stat(L) $\triangleleft$ dyn(R).

Note that because $\triangleleft$ generalizes inheritance, this technique saves many run-time type-checks that are inserted by SIMULA and BETA implementations.

10 Conclusion

We have presented a new approach to genericity in object-oriented languages. It has none of the drawbacks of parameterized classes and offers many pragmatic advantages: any class is generic, can be "instantiated" gradually without planning, and has all of its generic instances as subclasses.

Acknowledgements. The authors thank Peter Mosses and Ole Lehrmann Madsen 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] Luca Cardelli. Typeful programming. Technical report, Digital Equipment Corporation, 1989.

[6] W. R. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

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

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

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

[10] 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.

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

[12] J. D. Ichbiah et al. *Reference Manual for the Ada Programming Language.* US DoD, July 1982.

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

[14] 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.

[15] 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.

[16] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications.* Sigplan Notices, 21(11), November 1986.

[17] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Scaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[18] 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.

[19] Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen. Strong typing of object-oriented languages revisited. In *Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming*, 1990.

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

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

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

[23] 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.

[24] Jens Palsberg and Michael I. Schwartzbach. Genericity And Inheritance. Computer Science Department, Aarhus University. In preparation.

[25] U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 289–297. ACM, 1988.

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

[27] 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.

[28] 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.

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

[30] 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.

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