Strong Typing of Object-Oriented Languages Revisited

Ole Lehrmann Madsen
Boris Magnusson
Birger Møller-Pedersen

LU-CS-TR:90-66

Also published in Proceedings of OOPSLA/ECOOP'90
Conference on Object-Oriented Programming: Systems, Languages, and
Applications, SIGPLAN Notices, vol. 25, no. 4, pp. 140, 1990

Department of Computer Science
Lund University

P.O. Box 118
S-221 00 Lund
Sweden

Strong Typing of Object-Oriented Languages Revisited

Ole Lehrmann Madsen

Computer Science Department, Aarhus University

Ny Munkegade, DK-8000 Aarhus C, Denmark

Tlf.: +45 6 12 71 88 - E-mail: olmadsen@daimi.dk

Boris Magnusson

Department of Computer Science, University of Lund

PoBox 118, S-221 00 Lund, Sweden

Tlf.: +46 46 10 80 44 - E-mail: boris@dna.lth.se

Birger Møller-Pedersen

Norwegian Computing Center

P.O. Box 114, Blindern, N-0314 Oslo 3, Norway

Tlf.: +47 2 45 35 00 - E-mail: birger@nr.no

November 1, 1993

Abstract

This paper is concerned with the relation between *subtyping* and *subclassing* and their influence on programming language design. Traditionally subclassing as introduced by Simula has also been used for defining a hierarchical type system. The type system of a language can be characterized as *strong* or *weak* and the type checking mechanism as *static* or *dynamic*. Parameterized classes in combination with a hierarchical type-system is an example of a language construct that is known to create complicated type checking situations. In this paper these situations are analyzed and several different solutions are found. It is argued that an approach with a combination of static and dynamic type checking gives a reasonable balance also here. It is also concluded that this approach makes it possible to base the type system on the class/subclass mechanism.

1 Introduction

The purpose of this paper is to contribute to the clarification of typing issues in object-oriented languages. The issue of type checking in languages that has *classes with type parameters* have recently been the subject of several papers [2, 10, 11]. There has also been proposals for introducing a separate type system supplementary to the subclass hierarchy [1].

We will investigate these problems from the point of view taken in the Scandinavian school of object-oriented programming [4, 8, 12]. The class concept was introduced in Simula [3] and its motivation was to model concepts in the application domain. This lead to the introduction of subclassing mechanisms as a means to represent specialization and generalization hierarchies. Inheritance of properties in these hierarchies was the main motivation to introduce subclassing. Viewed as a modeling mechanism, subclassing was also taken to define a hierarchical type system. Types are defined explicitly which means that separate classes with the same internal structure define different types. This approach has been followed in Smalltalk [5], Beta [7], C++ [15] and Eiffel [10].

Apart from being used for (1) modeling as originally intended, subclassing has also been used as a means of specifying (2) inheritance of code or "code sharing". A type system can also be viewed in two different ways (1) as a means for representing concepts in the application domain and (2) as a means for detecting certain kinds of program errors, type errors. The first aspect, representation of concepts, is covered by both mechanisms and subclassing has been used also to define a hierarchical type system which have thus made a separate type system unnecessary.

The strength of the type system is also a related issue. We regard this as a continuum where *weakly typed* means that the type of an expression carries little or no information. Smalltalk is an example of such a language where the type of instance variables convey very little information on what messages are legal to send to the denoted object. A perfectly *strongly typed* language would exclusively have expressions where its type carries all information about the denoted object. We are not aware of any such object-oriented language although some other languages come close [16]. Languages with a hierarchical type system and qualified references serve as a compromise since some, but not necessarily all, operations on an object can be inferred from the qualification of the reference. Strong typing also implies that type checking can be performed during compilation, but utilization of weak typing aspects has to be checked during run-time.

Weak typing and run-time type checking provides a greater flexibility for the programmer. This flexibility is especially appreciated when developing the initial prototypes of a system. Compile-time type checking is useful for three purposes: (1) it improves the readability of programs (2) it makes it possible to detect a certain class of program errors at compile-time and (3) it makes it possible to generate more efficient code. A programming language must offer a proper balance between flexibility and compile-time checking to be useful.

Most so-called strongly typed languages rely on a combination of compile-time type checking and run-time type checking. Simula and BETA are examples of such languages. Here a large class of type errors are caught at compile-time, whereas others are left to run-time checks. C++ is an example utilizing compile-time type checking to some extent, but situations that can not be caught during compilation are not cared for during run-time. Eiffel is an example of a language which has been designed to permit a high degree of compile-time type checking, but there are, however, also situations where run-time type checking has to be used.

Parameterized classes in combination with a hierarchical type- system are known to create complicated type checking situations [2, 10, 11]. Similar situations can be created with a variety of language constructs in most object-oriented languages. This paper is using the type system of BETA, which extends the type system of Simula, for illustrating the problems. The extension allows for classes parameterized with classes (types) by means of so-called virtual classes as described in [9]. The type checking problems that arise in such situations are analyzed and alternative solutions are described. It is argued that an approach with a combination of static and dynamic type checking gives a reasonable balance also here, again avoiding a separate type system. Language constructs which can be used to reduce the amount of run-time checks in certain situations are also discussed.

In addition some of the examples from [1] are shown in BETA. The notions of qualified references, remote access and reference assignment as described in section 2-4 are the same in BETA and Simula.

The language notation used in this paper is a modified version of BETA similar to the notation used in [9].

2 Qualified references

Consider the following class hierarchy:

```
Vehicle: class (# owner: @integer;
                  licenseNo: @integer
               #);

Bus: class
     Vehicle (# noOfSeats: @integer;#);

Truck: class
       Vehicle (# tonnage: @integer  #);

Car: class Vehicle (# #);

aVehicle: ^ Vehicle;
aBus: ^ Bus;
aTruck: ^ Truck;
```

The classes `Bus`, `Truck`, and `Car` are subclasses of class `Vehicle`. The attributes `owner`, `licenseNo`, `noOfSeats`, and `tonnage` are local integer variables, while `aVehicle`, `aBus`, and `aTruck` are references (or pointers) to objects.

A reference is *qualified* by a class. The qualification restricts the set of objects that the reference may refer to. The reference `aVehicle` is qualified by *Vehicle*. This implies that `aVehicle` may denote instances of class `Vehicle` and instances of subclasses of `Vehicle`. The reference `aBus` may denote instances of class `Bus` and instances of possible subclasses of `Bus`. It is, however, not possible for the reference `aBus` to denote instances of class `Vehicle`, `Truck`, and `Car`.

In order to describe the type system more precisely we will introduce some formal, but yet very simple notation.

The class hierarchy in Beta (and other object-oriented languages) can be described as a lattice since the classes are partially ordered by the relation *subclass of* or the symbol $\subset$ (the relation $\supset$ is used for *superclass of*). A pre-defined class with the property of being the superclass of all other classes are sometimes explicitly available in the language (class `Object` in Smalltalk and Beta). In the lattice this class plays the role of Top.

Object

Vehicle

Bus Truck Car

NoClass

The special class `NoClass` is a subclass of all other classes. It plays the role of bottom in the Lattice. A reference that refers to no object has the value `NONE`. The class of `NONE` is `NoClass`. Class `NoClass` is introduced for purely technical reasons.

We also introduce two functions with the following definitions: **object()** returns the object that a reference actually denotes, **qual()** returns the formal class of a reference or the actual class of an object.

The declaration

```
R: ^ T
```

implies that $qual(R) = T$. This means that **qual(**`R`**)**, where `R` is a reference, may be computed at compile-time. If `X` is an object, then **qual(**`X`**)** is also constant. Consider

```
new T[] -> R[]
```

The action `new T` generates an object `X` such that $qual(X) = T$.

The purpose of `NoClass` is to make sure that $qual(object(R))$ is well defined for any reference `R`. For the value `NONE` we have $qual(NONE) = NoClass \subset T$ for any class `T` which is not `NoClass`.

The function **object(**`R`**)** varies at run-time, since `R` may denote different objects. The role of the function **object()** is to express dependency on run-time behavior.

The idea of qualified references can now be stated with the following relation that always must be true for all references in a program:

$$qual(object(ref)) \subseteq qual(ref)$$

As an example consider the reference `aVehicle`.The relation

$$qual(object(aVehicle)) \subseteq qual(aVehicle)$$

expresses the same restriction on what objects the reference `aVehicle` may denote as expressed above in English.

3 Remote access of attributes

The qualification of references is used to check at compile-time that remote-access of attributes is legal. The following remote-identifiers are legal:

```
aVehicle.owner;
aBus.owner;
aBus.noOfSeats;
aTruck.owner;
aTruck.tonnage;
```

As an example the first remote access is legal since $qual(aVehicle) = Vehicle$ and the class `Vehicle` specifies an attribute `owner` and so on. The following remote-identifiers are illegal:

```
aVehicle.noOfSeats;
aBus.tonnage;
```

In these simple examples the attributes have all been instance variables, but procedures may also be attributes with the same rules for legal attribute access. Remote access of procedure attributes correspond to message sending in Smalltalk. In Smalltalk reference variables are not qualified, so it is not possible to check at compile-time that a message is legal. Assuming that `noOfSeats` is a method attribute, a Smalltalk expression like

```
aVehicle noOfSeats
```

will give rise to the run-time error: "Message not understood". The qualification of references makes it possible to check at compile-time that this kind of error does not occur. There is still the possibility that a reference may be NONE, which gives rise to an error, but this is not considered a type-checking problem.

4 Reference assignment

Qualified references provide less flexibility than unqualified references which may denote objects of any class. The cost of this flexibility is *a run-time check for each message sending*. The comparable cost for qualified references is significantly smaller: *A run-time check that will take place at some cases of reference assignment.*

Consider the following example:

```
new Bus[] -> aBus[];          {1}
aBus[] -> aVehicle[]          {2}
aVehicle[] -> aBus[]          {3}
aTruck[] -> aBus[]            {4}
```

In {1} a new instance of `Bus` is generated. The instance reference is assigned to the reference `aBus`. This statement is trivially legal and this can be checked during compilation. From

$$qual(object(new\ \ Bus)) = Bus$$

and

$$qual(aBus) = Bus$$

it follows that

$$qual(object(newBus)) \subseteq qual(aBus)$$

This ensures that

$$qual(object(aBus)) \subseteq qual(aBus)$$

holds true after the assignment.

The assignment in {2} is legal since `aVehicle` may denote `Bus` objects. The legality may be (and is) determined at compile-time, i.e. no run-time checking will take place. The legality can be statically checked since

$$qual(object(aBus)) \subseteq Bus$$

and

$$Bus \subseteq qual(aVehicle) = Vehicle$$

The assignment in {3} is legal if `aVehicle` denotes an instance of `Bus`. In general this may not be detected at compile time. This implies that a run-time type check will be performed in this case. This can be seen since the static information

$$qual(object(aVehicle)) \subseteq Vehicle$$

cannot be used to guarantee that

$$qual(object(aBus)) \subseteq Bus$$

is true after the assignment. The compiler must thus emit a run-time check to ensure that

$$qual(object(aVehicle)) \subseteq Bus$$

The assignment in {4} is illegal, since it is not possible for `aTruck` to denote a `Bus` object. This is statically checkable since

$$qual(object(aTruck)) \subseteq qual(aBus)$$

give rise to comparison between `Truck` and `Bus` and these two classes are not ordered.

In general an assignment statement `A[]->B[]` (`B:-A` or `B:=A` in other languages) must be checked to fulfill

$$qual(object(A)) \subseteq qual(B)$$

in order to ensure that

$$qual(object(B)) \subseteq qual(B)$$

holds after the assignment.

If $qual(A) \subseteq qual(B)$ this is statically correct, but if $qual(A) \supset qual(B)$ then the compiler must ensure that $qual(object(A)) \subseteq qual(B)$ at run-time.

In most cases assignments are as in {2} (or qualifications are equal) and no run-time checking is needed. The ability to weaken the type information on an object as in {3} is very usable in order to write general code like queue and list manipulation etc. The problem of managing a queue of Vehicles is described already in [4]. The result of a queue operation, like returning the first object, is such a weakly qualified reference. The possibility to explicitly strengthen the type information again (as in {3}) is vital. This gives the programmer the possibility to view an object at different levels of abstraction. In the example above one would typically use the following views: as an element in a queue, as a Vehicle, and as a Bus or Truck.

Consider the example of three registers: A Vehicle register, a Bus Register, and a Truck register. The Vehicle register may contain references to both Bus objects and to Truck objects. The qualification of references used to build up the Vehicle register will therefore be `Vehicle`. The parameter to an insert operation on the Vehicle register will be qualified by `Vehicle`, and operations for getting references to objects in the register will deliver references qualified by `Vehicle`. Similarly the Bus register will use references qualified by `Bus`. The task of extracting `Bus` objects from the general Vehicle register and entering them into the Bus register will necessarily involve an assignment like that of {3}.

In Simula, BETA and C++ this kind of assignment is possible. Release 2.2 of Eiffel offers a new assignment operator. The assignment in {3} will be `aBus ?= aVehicle` in Eiffel. This operator assigns the object to `aBus` if legal according to the rules given above, NONE otherwise. The execution of this operator thus gives rise to an implicit run-rime check.

5 Value assignment

In this section the notion of *value assignment* is handled. Rules similar to those for reference assignment apply, and the same kind of run-time type checking apply.

By value assignment is meant some form of copying of state of the objects. The exact way of copying differs from language to language. For the purpose of this paper we will assume that value assignment means copying the state of the source object to the state of the destination object. We introduce two functions: **copy**(`r1,r2`) copies the contents of `r1` to `r2` assuming that **qual**(`r1`)=**qual**(`r2`), while **project**(`r,qual`) selects the **qual**-part of `r`. The variables `r1, r2` and `r` all denote objects.

Value assignment in BETA has the form:

```
aBus1 -> aBus2
```

The assignment above will have the effect that the values of the attributes `owner`, `licenseNo` and `noOfSeats` of `aBus1` are copied to the corresponding attributes of `aBus2`, that is

$$copy(aBus1, aBus2)$$

We are here assuming that `aBus1` and `aBus2` are qualified by `Bus`, that is

$$qual(aBus1) = qual(aBus2) = Bus$$

Consider the following value assignment:

```
aBus -> aVehicle  {1}
```

The reference `aBus` refers to an object that is at least a `Bus` object and `aVehicle` refers to an object that is at least a `Vehicle`. Only the `Vehicle` attributes of the object denoted by `aBus` may be copied to the object denoted by `aVehicle`. This is stated by the expression:

$$copy(r1, r2)$$

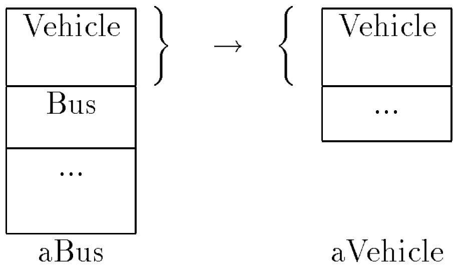
where

$$r1 = project(object(aBus), Vehicle)$$

and

$$r2 = project(object(aVehicle), Vehicle)$$

The following picture illustrates the situation. Only the "`Vehicle` bits" of the object referred to by `aBus` are copied to the corresponding "`Vehicle` bits" of the object referred to by `aVehicle`. The dots ... indicates that these objects may in fact be "larger" than can be deferred from the formal qualification of their references.



Consider next an assignment of the form:

```
aVehicle -> aBus  {2}
```

The situation here is opposite to the previous situation. A potential "smaller" object is copied to a potential "larger" object. At compile-time it is known that the two objects at least have the `Vehicle` attributes in common. Another possibility would be to require that `aVehicle` denotes an object qualified by `Bus`. (This would be analogous to the situation with reference assignments of the form `aVehicle[] -> aBus[]`.) In this case the `Bus` attributes could be copied.

With the above semantics of value assignment it is determinable at compile-time which bits to copy from the source object to the destination object. This is essentially the semantics of value assignment in BETA.[1] This semantics has the disadvantage that in some situations some information (bits) may be lost. Consider case {1}: If `aVehicle` actually refers to a `Bus` object, then all the attributes of the `Bus` object referred to by `aBus` could be copied to the `Bus` object referred to by `aVehicle`. In general one could find the largest common subclass of the actual objects being denoted and then copy the common attributes.

In most languages value assignment is defined as a pure copying of bits. In this way the state of one object may be forced upon another object. Often different object states may denote the same abstract value. It is therefore not always desirable to define the semantics of value assignment as a bitwise copying. The situation is even worse when considering *equality*. Here a bitwise comparison of two objects may not correspond to equality of the abstract values represented by the two objects. In general it is difficult to suggest language constructs for handling value assignment and equality.

Finally we notice that in most work on hierarchical type-systems the distinction between reference semantics and value semantics of assignment and equality does not seem to be explicit. In relation to object-oriented programming this distinction is, however, crucial.

[1] See, however, [7] for a more precise description of value assignment

6 Classes with "type" parameters

Virtual classes in BETA and generic classes in Eiffel makes it possible to define classes parameterized with other classes or types. These are very powerful language mechanisms, but they also complicate the rules for checking the legality of assignments.

The following examples show that subclassing and the rules for assignment associated with subclasses may well be used for expressing types of parameters. We will use it as basis for the analysis. The discussion will be generalized at the end of the section.

Consider a procedure for entering any vehicle into some kind of register. As part of entering, the vehicle is going to get a license number (the owner of the vehicle is supposed to be given). `newLicenseNo` is a function that delivers a new integer value:

```
Insert: proc
  (# par: ^ Vehicle
  enter par[]
  do newLicenseNo -> par.licenseNo;
     {enter par into register}
  #)
```

As the parameter `par` is known to denote instances of at least class `Vehicle`, it is safe to access the `licenseNo` attribute.

If `aBus` denotes an instance of class `Bus`, that is a subclass of `Vehicle`, which is the "type" of the parameter, then the following invocation of `Insert` should be legal:

```
aBus[] -> Insert
```

Type legality of this is handled by the rule for reference assignment (from `aBus[]` to the `par[]` parameter of `Insert`).

Consider the `Insert` to be part of a general `Register` class, parameterized by the type of vehicles to be in the register. The intention is to define specialized `Registers` restricted to hold objects of class `Bus` (and its subclasses), and similar specialized registers for `Truck` and `Car` objects. The qualification, `Type`, of the parameter, `par`, to the procedure `Insert` is declared virtual. This is done in order to strengthen its qualification in specializations of `Register`.

A general `Register` is able to hold any vehicle, i.e. references to objects of the classes `Bus`, `Truck`, `Car` and also `Vehicle` objects.

```
Register: class
   (# Type: virtual class Vehicle;
      Insert: virtual proc
         (# par: ^Type
         enter par[]
         do ...;
            newLicenseNo->par.licenseNo;
            ...; INNER; ...
         #);
   #);
```

Given this general class the following subclass of `Register` is a class of registers that may only hold buses. This is accomplished by restricting the virtual class `Type` to `Bus`:

```
BusRegister: class Register
  (# Type: extended class Bus;
     Insert: extended proc
       (#
       do (par.noOfSeats,par.owner[])
            ->checkValidity;
          INNER
       #)
  #);
```

Restricting the type `Type` to `Bus` serves two purposes. It enables us to say that the parameter to the procedure `Insert` must be at least a `Bus` which will then guarantee that non-`Bus`es will not be inserted (there is most likely also some internal data structures in the `Register`, not shown in the example, where the same restriction apply). The problem of how to enforce this restriction is the topic of the rest of this section. The second purpose is that inside the `BusRegister` it will be safe to access attributes of class `Bus`. The expression `Par.noOfSeats` is an example of that. This add to the expressiveness of the language and can be statically type-checked.

The qualification of a virtually qualified reference is in general not known at compile time since the qualification can be extended to a more specific class. In our example `par` is qualified `Vehicle` in a `Register`, but as `Bus` in a `BusRegister`. For a reference like `par` qualified by a virtual class we have the following relation:

$$qual(par) = Type \subseteq Vehicle$$

Since `Type` may have different extensions in different subclasses of `Register`, we cannot determine $qual(par)$ at compile-time.

The qualification `aRegister.Type` depends on the qualification of `aRegister`. We have the following assertions:

$$aRegister.Type = Vehicle$$
$$if\ qual(object(aRegister)) = Register$$

and

$$aRegister.Type = Bus$$
$$if\ qual(object(aRegister)) = BusRegister$$

In general each subclass of `Register` gives rise to an assertion of this kind. We use the notation

$$object(aRegister).Type$$

to denote this virtual qualification.

The qualification of the parameter `par` also depends on the qualification of `aRegister`. We use the notation

$$qual(object(aRegister).Insert.par)$$

to denote the qualification of a specific `par`.

We can deduce that

$$object(aRegister).Type \subseteq Vehicle$$

and

$$qual(object(aRegister).Insert.par) \subseteq Vehicle$$

but these relations are of little practical use when determining the legality of assignments to `par` as will be seen below.

For an assignment

```
aVehicle[] -> aRegister.Insert[]
```

we must prove that after the assignment the following holds:

$$\begin{aligned} &qual(object(aRegister.Insert.par)) \\ &\quad \subseteq qual(object(aRegister).Insert.par) \end{aligned}$$

This relation states that the qualification of the object referred to by `par` must be a subclass of the qualification of `par` associated with the object actually referred to by `aRegister`. Since the left-hand side of this relation is $\subseteq Vehicle$ we can conclude that the demand on the object referred to by `aVehicle` is at least a `Vehicle`, but it may be stronger.

In the following we analyze three situations of assignment to virtually qualified references. In case 1 we consider an assignment independent of its context. In cases 2 and 3 we take the context of the assignment into consideration. In all 3 cases we assume the following declarations:

```
aVehicle: ^Vehicle;
aRegister: ^Register;
```

Case 1: Consider the assignment

```
aVehicle[] -> aRegister.Insert;  {1}
```

The problem here is that the qualification of $object(aRegister)$ cannot be determined at compile-time. If

$$qual(object(aRegister)) = Register$$

then

$$aRegister.Type = Vehicle$$

and the assignment {1} is legal.

If on the other hand

$$qual(object(aRegister)) = BusRegister$$

then

$$aRegister.Type = Bus$$

and the assignment {1} is legal only if

$$qual(object(aVehicle)) \subseteq Bus$$

This implies that in the general case assignments to virtually qualified references can not be statically checked. Given more information from the program it is in principle possible to calculate an upper bound on the needed qualification. This is done in the next two cases.

Case 2: Consider the imperatives:

```
new Register[] -> aRegister[];
new Bus[]  -> aVehicle[];
aVehicle[] -> aRegister.Insert;  {2}
```

Here we have:

$$qual(object(aRegister)) \;=\; Register$$

$$\begin{aligned} & qual(aRegister.Insert.Par) \\ & \quad = aRegister.Type \\ & \quad = Vehicle \end{aligned}$$

$$qual(aVehicle) \;=\; Bus$$

and the reference relation will be fulfilled since $Bus \subset Vehicle$.

Case 3: Consider the imperatives:

```
new BusRegister[] -> aRegister[];
new Vehicle[] -> aVehicle[];
aVehicle[] -> aRegister.Insert;  {3}
```

Here we have:

$$qual(object(aRegister)) \;=\; BusRegister$$

$$\begin{aligned} & qual(BusRegister.Insert.par) \\ & \quad = BusRegister.Type \\ & \quad = Bus \end{aligned}$$

$$qual(aVehicle) \;=\; Vehicle$$

so the reference relation is violated in this case.

The examples 1-3 shows that the assignment can not be statically type checked in the general case. If the compiler can infer the type of an object using dataflow analysis it can statically check both 2 and 3. Full dataflow analysis is not possible, but a limited form, only targeted to recognize the case when a reference can be guaranteed to have exactly its declared qualification has been proposed for Eiffel [11]. This effect can also be achieved in Beta with part objects that are statically allocated. (See below). In [13] the same effect is achieved for so-called homogeneous variables which are type exact.

The type checking problem described above is general and occurs in a couple of different language constructions. The use of virtual qualification was chosen to illustrate the problem above. Classes with type parameters shows the same problem when subclasses are allowed to strengthen the qualification on the type parameter. The same effect can also be achieved with self-relative types as "like Current" and "thisClass". Yet another example is classes with virtual procedures. If subclasses are allowed to restrict the type of the parameters to re-implementations of procedures the same problem occurs again.

The heart of the problem can be explained by observing that the notion of qualified references does not help us in this case. The essence of qualified references is to guarantee that a reference is denoting at least an object of a certain class. This is useful because it is then safe to assume that the object has attributes of that class. When assigning an object to such a reference the compiler need to calculate the qualification of the reference as described in section 4, and the legality of the assignment can in many cases be checked statically.

Qualified references allows us to determine a least qualification of an object, but the group of constructions described earlier in this section introduces objects where the demands may increase with the qualification. The notion of qualified references can not help us to calculate an upper bound on these demands. We have found the following three different ways to safely handle type-checking of type-parameterized constructions:

1. Not allowing the type-demands to be strengthen

2. Introducing references that are type exact

3. Run time checks

The first solution has been adopted for example in Simula for virtual procedures with specified parameters and for arrays of references used as parameters which must conform exactly. For these situations it works also in practice due to the possibility of dynamically strengthening the qualification of an object. This solution have also been proposed in [2] where it is suggested that it should not be allowed to strengthen the type demands in subclasses, but only weakening them. Weakening type-constraints is also possible in Trellis/Owl [14], but seems to be of very limited practical value and will in practice mean that a fixed type will be used.

In BETA a virtual class can be fixed in a subclass (see below) with the meaning that a declaration can not be further strengthen in a subclass. [2] suggests the technique with weakening the demands to be used also for classes with type parameters. Also here weakening is of questionable practical value. It should be noted that BETA offers this as an alternative, while [2] suggest this to be the only alternative.

The second solution is exemplified with part objects in Beta and the suggested type-enforce rule in Eiffel. One can also consider to introduce a new kind of references which always denote objects belonging to exactly the declared class.

The choice is between expressive power and statically type-checkable constructions. In Beta the choice has been to allow also constructions that require run-time type checking. This route has been followed here, but also for dynamic strengthening of qualification as described in section 4.

Limiting run-time checks

In general the use of virtual classes will involve run-time checking. BETA has constructs that makes it possible to avoid some of these run-time checks. In class `BusRegister`, the virtual class `Type` may be defined using `fixed` instead of `extended`. This implies that `Type` is not a virtual class in `BusRegister`, so it cannot be further extended in subclasses of `BusRegister`.

The same effect can be obtained by declaring a static (part) object like:

```
aFixedBusRegister: @BusRegister
```

Here it is also known that no further extension of `Type` is possible, so the qualification of `par` in `Insert` is fixed to the class `Bus`. The reason is that virtual classes and procedures may only be extended in subclasses, and `aFixedBusRegister` is not a class, but an object. Note that `aFixedBusRegister` is in fact a type exact reference.

7 Subclassing versus subtyping

In [1] it is claimed the need for a special interface inheritance hierarchy that is different from the class/subclass hierarchy, and that the interface hierarchy should be used for type checking purposes (and not the class/subclass hierarchy).

The following example is the BETA version of some of the examples from [1], and it demonstrates that it is possible to use the class/subclass hierarchy for type checking. As the previous examples in this paper it also introduces the need for run-time type checking.

Even though the BETA approach is to use the class/subclass hierarchy for type checking, this is not the same as to say that we do not want to distinguish between interface and implementation of a class. The language has a separate mechanism for that [6], but this will not be covered here.

The example demonstrates that it is possible to let a `ColorPoint` be a subclass of `Point`, and still have procedures local to `Point`, such as e.g. `Equal`, also work for objects of the subclass.

```
Point: class
  (# X,Y: @integer;

     Move: virtual proc
       (# dx,dy: @integer;
          P: ^ThisClass
       enter(dx,dy)
       do new ThisClass[] -> P[];
          x + dx -> P.x; y + dy -> P.y;
          INNER
       exit P[]
```

```
        #);
      Equal: virtual proc
        (# P: ^ThisClass;
           eq: @boolean
        enter P[]
        do ((x=P.x) and (y=P.y)) -> eq;
           INNER
        exit eq
        #)
    #);
```

In the same way as the language contains the special reference expression `ThisObject` giving the object in which the expression is evaluated, it also makes use of the pseudo-name `ThisClass`. The class `ThisClass` is the class of `ThisObject`, i.e.

$$ThisClass = qual(object(ThisObject))$$

In order for the parameter `P` of `Equal` to work not only for `Point`, but also for subclasses of `Point`, it is qualified by `ThisClass`. In a `Point` object the qualification `ThisClass` is `Point`. Consider the subclass `ColorPoint` of `Point`:

```
ColorPoint: class Point
  (# c: @color;

     Move: extended proc
       (# do c -> P.c; INNER #);
     Equal: extended proc
       (#
       do (eq and (c=P.c)) -> eq;
          INNER
       #)
  #)
```

In objects of the subclass `ColorPoint` the `ThisClass` is `ColorPoint`. This implies that the qualification of the parameter `P` is then `ColorPoint`, so `Equal` may also be extended to test for the equality of the `c` attribute of `ColorPoint` objects. This can also be expressed as:

$$qual(object(P)) \subseteq qual(object(ThisObject))$$

It should be noted that the notion of `ThisClass` may be obtained as a virtual class. In `Point` it would be defined as a virtual class, e.g.

```
thisClass: virtual class Point
```

and in `ColorPoint` it would be extended by

```
thisClass: extended class ColorPoint
```

See [9] for a further discussion of this.

The above example illustrate the same problem as discussed in section 6. This time the construction `ThisClass` is the cause of qualification strengthening. The following examples of use of the classes will illustrate the type-checking problems.

```
Pl,Pr: ^ Point;
Cl,Cr: ^ ColorPoint

Pl[]->Pr.Equal;          {1}
Cl[]->Cr.Equal;          {2}
Cl[]->Pr.Equal;          {3}
Pl[]->Cr.Equal;          {4}
```

In all these four cases there has to be inserted run-time tests analogous to that of the example in section 6. Assuming that `Pr` and `Pl` actually denote `Point`s while `Cl` and `Cr` denote `ColorPoint`s only the last case will actually fail during run-time. This is because we try to compare a `Point` object and a `ColorPoint` object by executing the `Equal` procedure of the `ColorPoint` object with the `Point` object as the parameter `P`. As pointed out in [1] this would lead to evaluation of the expression `P.c`, with `P` denoting a `Point` object, and this is invalid because a `Point` object does not have an attribute `c`.

The following is examples of situations where run-time checking may be avoided by using part objects. Suppose that the following objects are given:

```
P1,P2: @Point;
C1,C2: @ColorPoint;

P1[]->P2.Equal;          {5}
C1[]->C2.Equal;          {6}
C1[]->P1.Equal;          {7}
P1[]->C2.Equal;          {8}
```

Here it is known at compile-time that `P1` and `P2` refer to instances of `Point` and that `C1` and `C2` refer to instances of `ColorPoint`. In other words `P1,P2,C1` and `C2` are constant references. The effect is that case 5-8 can be statically type-checked. Case 5-7 will pass but in case 8 there will be found a type-error.

Recalling the three different solutions to this type-checking problem we will find the following:

1. **Not allowing the type-demands to be strengthened.**

   Adopting this attitude, the definition of class `ColorPoint` is wrong since it is strengthening the demands on the parameter `P` of the procedure `Equal` (and `Move` as well). This is the attitude taken in [1]. It has the effect that the above and many other programs will be illegal. In [1] it is phrased slightly different, the two classes `Point` and `ColorPoint` are found not to be type compatible.

2. **Introducing references that are type exact.**

   The effect of this possibility is shown above using part objects. All the expressions above are statically checkable. If this is the only alternative we can not write general code managing ColorPoints (and possibly many other sub classes) as Points which is of great practical value. This is also the effect of the suggested restriction for Eiffel [11]. The proposal in [13] for homogeneous variables is another example of this. Although type exact variables is a useful mechanism in many situations, we find it a too strong restriction to be the general case.

To conclude this discussion we finally also show the BETA formulation of one other example in [1].

```
Test: proc
   (# X,Y: ^ Point;
   enter(X[],Y[])
   exit X[]->Y.Equal    {a run-time check}
   #);
```

Since it is not statically known whether or not X and Y refer to instances of Point or ColorPoint it is necessary to perform a run-time check in the call X[]->Y.Equal. If X refers to a ColorPoint, then Y must also refer to an instance of ColorPoint or:

$$qual(object(X)) \subseteq qual(object(Y).Equal.P)$$

The procedure Test may be called in the following way:

```
(Pl[],Pr[])->Test   {1}
(Cl[],Cr[])->Test   {2}
(Cl[],Pr[])->Test   {3}
(Pl[],Cr[])->Test   {4}
```

In none of these cases there is a need for a run-time check at the call since the procedure Test only requires its arguments to be of class Point. The execution of Test will in all cases perform a run-time test when executing X[]->Y.Equal. In the fourth case the run-time test in procedure Test will fail since the class of X (Pl) is Point, while the the relation will demand at least a ColorPoint (again assuming that Pl is denoting a Point object).

Again using the first solution only case 1 will be accepted by the compiler since ColorPoint is not considered a subclass of Point. This would require the programmer to write one version of the Test procedure for each combination of argument types.

The second solution would lead to exactly the same situation since the pointers X and Y would be considered to have exact qualification match.

8 Conclusion

Issues regarding typing and programming language design have been the subject of this paper. Programming is regarded as modeling a real or imaginary part of world. From this point of view we conclude that the most important feature of the class mechanism is the ability to model concepts. Subclassing models specialization and inheritance of properties from the application domain. Used in this way, the class hierarchy defines a type system that is understandable in terms of the application domain.

A type system based on the class concept has been described together with a discussion of how strong typing can be supported in a flexible way. The strength of a type system is regarded as the amount of information conveyed with the type of an expression. This information can be used for compile-time type checking and early error reporting. Several examples are given of weakening and strengthening the type of an expression. It is argued that a certain amount of such flexibility is needed in order to support different levels of abstraction. Type strengthening expressions gives rise to run-time type checking.

The problems arising when classes are parameterized with types (i.e. classes) have been analyzed and it has been shown that the traditional approach with run-time checks in certain situations can also be used here. It has also been discussed how the amount of run-time checks can be decreased and even completely removed by introducing certain restrictions in the language (type exact references, forbidding type strengthening). Finally it has been argued that these restrictions can be very useful in many situations, but only allowing these restricted cases will hamper the expressiveness of the language.

Acknowledgement. We have received many useful comments from colleagues, students and the anonymous referees.

References

[1] P.S. Canning, W.R. Cook, W.L. Hill, W.G. Olthoff: *Interfaces for Strongly-Typed Object-Oriented Programming*, In OOPSLA'89, Object-Oriented Programming Systems, Languages and Applications, Sigplan Notices, Vol. 24, No. 10, Oct. 1989

[2] W. R. Cook: *A Proposal for Making Eiffel Type Safe*, In ECOOP'89, European Conference on Object-Oriented Programming, Cambridge University Press 1989.

[3] O.-J. Dahl and K. Nygaard: *Simula-67 Common Base Language* Publication S-22, Norwegian Computing Center, Oslo 70, 72, 84. Current version: Programming Language - Simula, Swedish Standard SS.63.61.14, ISBN-91-7162-234-9

[4] O.-J. Dahl and K. Nygaard: *The Development of the Simula Languages*, In *History of Programming Languages*, ed. R. W. Wexelblat, Academic Press, New York, 1981.

[5] A. Goldberg and D. Robson: *Smalltalk-80: The Language and its Implementation*, Addison-Wesley Publishing Company, 1983.

[6] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *Syntax Directed Program Modularization*, In: *Interactive Computing Systems* (ed. P. Degano, E. Sandewall), North-Holland, 1983

[7] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen: *The BETA Programming Language*, In: B.D. Shriver, P.Wegner (ed.), *Research Directions in Object-Oriented Programming*, MIT Press, 1987.

[8] O.L. Madsen and B. Møller-Pedersen: *What Object-Oriented Programming may be — and what it does not have to be* In ECOOP'88, European Conference on Object-

Oriented Programming, Lecture Notes In Computer Science, Vol. 322, Springer Verlag, 1988.

[9] O.L. Madsen and B. Møller-Pedersen: *Virtual Classes — A Powerful Mechanism in Object-Oriented Programming*, In OOPSLA'89, Object-Oriented Programming Systems, Languages and Applications, Sigplan Notices, Vol. 24, No. 10, Oct. 1989

[10] B. Meyer: *Object Oriented Software Construction*, Prentice-Hall 1988

[11] B. Meyer: *Static Typing for Eiffel.* Interactive Software Engineering Inc., July 2, 1989

[12] K. Nygaard: *Basic Concepts in Object Oriented Programming*, Sigplan Notices, Vol. 21, No. 10, 128–132 (October 1986).

[13] J. Palsberg. M. I. Schwartzbach: *Substitution Polymorphism for Object-Oriented Programming*, In OOPSLA/ECOOP'90, Object-Oriented Programming Systems, Languages and Applications, Sigplan Notices, 1990.

[14] C. Schaffert et. al: *An Introduction to Trellis/Owl*, In OOPSLA86, Object-Oriented Programming Systems, Languages and Applications, Sigplan Notices, Vol. 21, No. 11, Nov. 1986.

[15] B. Stroustrup: *The C++ Programming Language*, Addison Wesley, 1986.

[16] N. Wirth : *The Programming Language Pascal*, Acta Informatica 1, 1971, 35-63.