Supporting Class Evolution by Typing Inheritance

F. J. Hauck

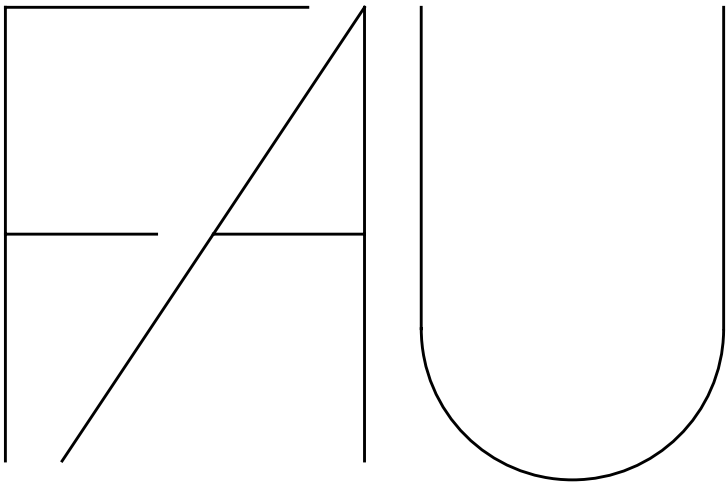
July 1993 TR-I4-5-93

Technical Report

Computer
Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



This paper was submitted and accepted as a position paper to the OOPSLA ‘93 Workshop W9 ‘Supporting the Evolution of Class Definitions’.

Supporting Class Evolution by Typing Inheritance

OOPSLA '93 Workshop W9 – Position Paper

Franz J. Hauck[1]

hauck@informatik.uni-erlangen.de
University of Erlangen-Nürnberg, IMMD 4
Martensstraße 1, P.O. Box 3429
D-91051 Erlangen, Germany

Abstract. While there are suitable type-systems for aggregation that allow the exchange of objects and classes, the configuration of systems, and dynamic binding, there are no type-systems for inheritance relationships. Thus, maintenance of class libraries becomes difficult or even impossible because changes in a class library can cause changes in the inheriting classes of an application program. Modeling inheritance with explicit and parametrical bindings introduces a typed interface for inheritance which allows type-safe changes in class libraries.

1 Introduction

An exchange of base classes is necessary for maintenance in class libraries, e.g. for bug-fixing. In this case application programmers should not be forced to change their programs when a library class is changed. Recompiling or rebinding of the application should be sufficient. This is only possible if there is a typed interface between base and subclasses. An improved base class in a library must be type-conforming to the inheritance interface of the prior class.

Using a typed interface for inheritance and modeling inheritance by explicit bindings allows an object to bind to an object of its base class at creation time. This binding can be procured by a name-server or broker similar to the dynamic bindings of aggregation relationships.

In chapter 2 we introduce a small object model upon which our approach is based. Aggregation and its typing are presented. Chapter 3 shows the properties of inheritance and introduces our approach to typed inheritance by modeling it with explicit bindings and aggregation. Chapter 4 concludes the results.

2 Object Model

2.1 Aggregation

An object consists of named variables[2]. A reference to another object (or to itself) can be bound to a variable by assignment. Bindings can be variable or constant. The composition of objects by bindings of references to variables is called *aggregation*.

In fig. 2.1 two objects *B* and *C* are bound to an object *A*, i.e. *A* has a reference to *B* and one to *C*. These bindings may be variable and changed at run-time. This corresponds to the concept of pointers in *C++* [1] and to variables in *Eiffel* [2]. Constant bindings are represented by object declarations in *C++* and by *expanded objects* in *Eiffel*.

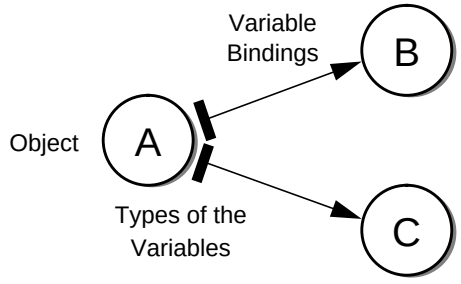


Fig. 2.1 Bindings between objects – Aggregation

Bindings can be established at the creation time of an object. Such bindings are called *initial*. Objects which are initially and constantly bound are called *private*.

1. This work is supported by the *Deutsche Forschungsgemeinschaft DFG* in the Sonderforschungsbereich *SFB 182 Project B2*.

2. These are often called *instance variables*.

2.2 Typing

In fig. 2.1 the variables are typed. The types are represented by thick cross-beams. Each object has a type. To bind an object to a variable, the object must have a type which conforms to the type of the variable. The typing of objects and variables allows a type-safe exchange of objects by rebinding. A new object has only to have a conforming type. For type-conformance the contravariant type-rule applies [3]. Informally a type *A* conforms to a type *B* when an object of type *A* can be used in every context where an object of type *B* can be used.

Types have their own representation in our object model and class-based inheritance is independent of subtyping. This decision is motivated in [4]. Class-based inheritance allows code-reuse. A type-based inheritance mechanism allows subtyping and "type-code"-reuse. The separation of classes and types needs a mechanism for combining both. Therefore, each class decides of which type it is.

3 Inheritance

Inheritance is class-based inheritance in the context of this paper. Type-based inheritance is out of the scope of this paper. In our object model class-based inheritance is modeled by a special aggregation relationship [5]. For a better understanding we demonstrate this by an example:

Let us think about a class *port*, whose objects have the ability to service a hardware port. We can output some characters to the port. Therefore, the class defines a method named *Putchar*. For the output of lines another method named *Putline* is defined. *Putline* is implemented by successive invocations of *Putchar* to output every single character of a line. To build a class which realizes a buffered output behavior for a hardware port we want to use inheritance. The class *bufferedport* redefines and replaces the method *Putchar* with a buffering version. All characters are inserted in a buffer. When the buffer is full an additional method *Flush* is called which invokes the original *Putchar* method to flush all the characters of the buffer. Therefore, the keyword *super* is used. The invocation of the method *Putline* calls the method of class *port* and the subsequent *Putchar* invocations in *Putline* are directed to the new buffering method of class *bufferedport*.

In fig. 3.1 a strongly simplified description of the classes *port* and *bufferedport* are given in a fictitious syntax. The keyword *self* is added to each method call for clarity. In most languages there is no need to write *self* explicitly at each call-statement.

```
port : class
{
   Putchar : ()->()
   { ... }

   Putline : ()->()
   {  ...
      while ...
         self.Putchar();
   }
}

bufferedport : class inherits port
{
   Putchar : ()->()
   {  ...
      self.Flush();
      ...
   }

   Flush : ()->()
   {  ...
      while ...
         super.Putchar();
   }
}
```

Fig. 3.1 Class for port objects – classic approach

In the chosen syntax, names of variables and methods are written on the left side of a colon, declarations or definitions on the right. Declarations of any formal parameters are left out for simplicity.

3.1 Inheritance by Aggregation

The example already gives some hints on how to model inheritance by aggregation. The keywords *self* and *super* are modeled by explicit variables. The variable *super* is bound to a private object of the base class. This variable is initially bound at creation time of the object of a subclass and the binding is constant[3]. The variable *self* is initially and constantly bound to the object of the subclass. Fig. 3.2 shows an object of class *B* in inheritance and aggregation relationships. In fig. 3.2a the bindings for a subclass *S* inheriting from *B* are shown[4].

3. Loosening this constraint could allow the modeling of dynamic inheritance relationships, but this is not a topic of this paper.

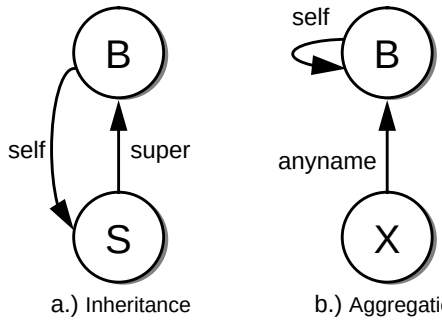


Fig. 3.2 Explicit binding of *self* and *super* in an inheritance and in an aggregation relationship

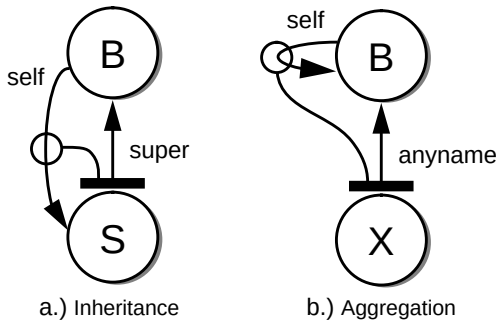


Fig. 3.3 Typed interface of the binding in an inheritance and in an aggregation relationship

Because the variable *self* is defined in the base class even when an object of the base class is not in an inheritance relationship, it is necessary to define how this variable is bound for the use of an object in an aggregation relationship only. For this case the variable is bound to the object itself. This situation is shown in fig. 3.2b.

The *self* variable of an object of the base class is variously bound depending on the aggregation or inheritance relationship the object is in. The binding is always initial and constant. Thus, it can be seen as a parameter for the creation of an object.

3.2 Typing

The typing of inheritance relationships has to cover simultaneously the bindings of the *super* variable of the subclass object and the *self* variable of the base class object. The type of the *super* variable describes the interface of the base class object. This type can be extended to describe the interface to inheritance. Therefore, the parametrical binding of the *self* variable is included into the type as a kind of publicly accessible variable. It is different from usual public variables because it is bound at creation time and constant for the remaining lifetime of the object.

Fig. 3.3 is the same as fig. 3.2 but shows the types of the variables *super* and *anyname* as a thick beam. The binding of *self* in the base class object is attached to the type of the *super* variable as a public variable, shown as a thin line from the beam to the binding arrow. Fig. 3.3a shows the object *S* and *B* in an inheritance relationship and fig. 3.3b sketches the objects *X* and *B* in an aggregation relationship.

```
port_t : type
{
   self : parm port_t;

   Putchar : ()->();
   Putline :()->();
}

port : class of type port_t
{
   self : parm port_t = here;

   Putchar : ()->()
   { ... }

   Putline : ()->()
   {  ...
      while ...
         self.Putchar();
   }
}
```

Fig. 3.4 Type *port_t* and class *port* with a parametrical binding of the variable *self*

Our approach can now be demonstrated by using the example from section 3.1. Fig. 3.4 shows the same example with a special aggregation relationship instead of inheritance. First of all a type for a port-object is described and named *port_t*[5]. The class *port* is defined to be of type *port_t*. The projection of names in the type to the names in the class definition is here implicitly done between methods and variables with the same name. Additional syntactical statements are necessary to allow explicit projections (renaming).

The class *port* defines an additional variable *self*. It is of type *port_t* which means that it can bind every object

4. For simplicity, in this figure only the binding for the *self* variable of the base class and not the *self* variable of the subclass is represented.

5. We name types according to an old C tradition with a suffix '_t'.

with a type conforming to *port_t*. The keyword *parm* indicates that the binding is constant and initial and can be set at the creation time of the object. The variable is part of the type. *Self* is defined with a default binding to the created object itself. Therefore, the keyword *here* is used to name the current object. Contrary to the traditional keyword *self* (e.g. in *Smalltalk*), *here* is not affected by inheritance, but always names the object in which it is used. The definitions of all other components of class *port* remain the same as in fig. 3.1.

```
var : port_t = port.new;
```

Fig. 3.5 A variable *var* bound to a new object of class *port*.

An object of class *port* can be created in an aggregation relationship as shown in Fig. 3.5. The *self* variable of the object is bound to the object itself according to the definition in the class *port*. This corresponds to fig. 3.3b.

```
bufferedport_t : type o> port_t
{
   self : parm bufferedport_t;

   Putchar : ()->();
   Putline : ()->();
   Flush : ()->();
}

bufferedport : class of type bufferedport_t
{
   self : parm bufferedport_t= here;
   super : priv port_t= port.new(self=self);

   Putchar : ()->()
   {  ...
      self.Flush();
      ...
   }

   Putline :- super.Putline;

   Flush : ()->()
   {  ...
      while ...
         super.Putchar();
   }
}
```

Fig. 3.6 Type *bufferedport_t* and class *bufferedport* using a private object of class *port* due to inheritance

For the class *bufferedport* we define a type *bufferedport_t* which conforms to the type *port_t*. This is defined using the operator ‘o>’ which means ‘conforms to’. The operator ‘<o’ means conformance in the opposite direction.

The type *bufferedport_t* conforms to *port_t* although it has a public variable *self* which has a different type than the corresponding *self* variable in *port_t*. This is possible because the types of *self* conform and the *self* variables are read-only. Variables can be seen as a pair of methods, one for reading and one for writing the variable. The read-method has an output-parameter, the write-method has an input-parameter of the variable-type. This causes a type covering a variable to be not conforming to a type covering a corresponding variable with a different type (may be a larger type). A read-only variable only has a read-method. Thus, the type *port_t* conforms to *bufferedport_t* even when the *self*-variables have different types.

In the class *bufferedport* we do not use a keyword to inherit from class *port*, but we define a *super* variable, which is bound to a private object of class *port*. Privacy is declared by the keyword *priv*. Within the parentheses the parametrical binding of *self* in *port* is set to the same object as *self* in *bufferedport* refers to[6]. This allows correct repeated inheritance from *bufferedport*. There, both *self* variables are changed by a parametrical binding.

Because objects of class *bufferedport* do not have a method with the name *Putline*, the name *Putline* is declared in *bufferedport* referring to the *Putline* method of the object bound to *super*. Therefore, the operator ‘:-’ instead of a colon is used. This operator defines an alias for the name at the right side.

The type *port_t* serves here as a type for the inheritance relationship between the classes *bufferedport* and *port*. It declares the parametrical binding of the *self* variable of *port* objects. The type of this variable indicates that in an inheritance relationship *port* objects call methods in the subclass just as they are defined in type *port_t*. Indeed *port* objects only call the *Putchar* method via *self*. A restriction of the type of *self* is possible and explained in section 3.4.

6. In the syntax of the setting of a parametrical binding (the assignment in parentheses) there has to be a name on the left side of the equal sign which is valid in the base class – here *self* of *port* – and there has to be an expression using names of the defining class on the right side – here *self* of *bufferedport*.

3.3 Type-Conformance for Inheritance

Improved or changed versions of class *port* have to conform to type *port_t*, but this conformance is not equivalent to the required conformance for typed aggregation relationships. The type *bufferedport_t* conforms to *port_t*, but the class *bufferedport* is not suited to replace class *port* in the inheritance relationship between *port* and *bufferedport*. This is because the setting of the parametrical binding is used at creation time of the base class object. This setting violates the requirement of an read-only variable. Thus we need two different relationships of conformance one for aggregation (*a-conformance*) and one for inheritance (*i-conforming*).

The type conformance rules for aggregation cover parametrical bindings, indicated by the *parm* keyword, as read-methods while type conformance rules for inheritance cover parametrical bindings as read- and write-methods. A class which enlarges the type of the *self* binding might be a-conforming but cannot be i-conforming Thus, a changed library class replacing an old one cannot change the type of *self*.

```
port2_t : type c> port_t
{
   self : parm port_t;
   self2 : parm port2_t;

   Putchar : ()->();
   Putline : ()->();
   Putint : ()->();
   Putfloat : ()->();
}

port2 : class of type port2_t
{
   self : parm port_t= self2;
   self2 : parm port2_t= here;

   ...
   Putint : ()->()
   {
      ...
      self2.Putfloat(...);
   }

   Putfloat : ()->()
   {
      ...
   }
}
```

Fig. 3.7 Type *port2_t* and class *port2*

Fig. 3.7 shows an extended version of class *port*, named *port2*. There are two methods, *Putint* and *Putfloat*, which are able to print integral and floating point numbers to the port. To complicate the example we realize *Putint* by converting integral numbers to floating point numbers and calling *Putfloat* for output. Thus, the type of *self* must cover at least the *Putfloat* method.

The implementation defines a type *port2_t* which is i-conforming to *port_t*. This is indicated by the operators ‘c>’ or ‘<c’ similar to ‘o>’ and ‘<o’. I-conformance includes a-conformance, too. Thus, objects of type *port2_t* can also be used in aggregation relationships typed with type *port_t*.

The class *port2* defines a parametrical binding *self* for all the old code taken from class *port*. A second parametrical binding called *self2* covers the new call of *Putfloat*. The result is a class which can be used instead of class *port* even in an inheritance relationship, a parametrical binding respectively. Then, only the *self* variable is set and the *self2* variable remains set on the default value *here*. Object creations using class *port2* with type *port2_t* set only the parametrical binding of *self2* which sets the binding of *self* accordingly.

3.4 Smaller Types for the *Self* Variables

As found at the end of section 3.2 the *self* variable in class *port* is defined with type *port_t*, but only the *Putchar* method is invoked via *self*. This shows that the type of the *self* variables could be smaller than the type of the class where they are defined. A larger type conforms to a smaller type.

```
selfport_t : type <o port_t
{
   Putchar : ()->();
}

port : class of type port_t
{
   self : parm selfport_t= here;
   ...
}
```

Fig. 3.8 Minimal type *selfport_t* for *self* in *port* and the changed class *port*

In our example we introduce a type *selfport_t* which covers the minimal (smallest) type for *self* in *port* (fig. 3.7). The type *port_t* is declared conforming to *selfport_t*. The *self* variable gets the new type. The type *port_t* has to be corrected in the same way.

```
selfbufferedport_t : type <o bufferedport_t,
                          o> selfport_t
{
   Putchar : ()->();
   Flush : ()->();
}

bufferedport : class of type bufferedport_t
{
   self : parm selfbufferedport_t= here;
   ...
}
```

Fig. 3.9 Minimal type *selfbufferedport_t* for *self* in *bufferedport* and the changed class *bufferedport*

The class *bufferedport* and the corresponding type must be changed accordingly. *Selfbufferedport_t* is declared conforming to *selfport_t* and *bufferedport_t* is declared conforming to *selfbufferedport_t*. These declarations allow the correct typing of the parametrical bindings of *super* and *self*. There are several statements in the code of the example in which the declared conformance relationships are needed. They are sketched in fig. 3.9.

port_t <o *bufferedport_t*
needed to use *bufferedport* objects as *port* objects

selfport_t <o *port_t*
needed for the assignment of *here* to *self* in *port*

selfbufferedport_t <o *bufferedport_t*
needed for the assignment of *here* to *self* in *bufferedport*

selfport_t <o *selfbufferedport_t*
needed for the parametrical binding of *self* in *port* to *self* in *bufferedport*

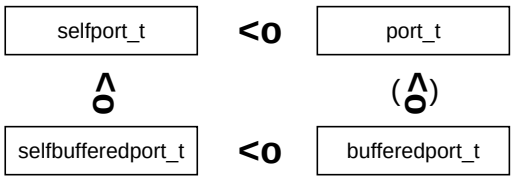


Fig. 3.10 Conformance relationships between the introduced types

We can argue that the conformance relationships sketched in fig. 3.9 must be valid for all inheritance relationships. The types of the *self* variable are smaller or equal to the type of the object in which they are defined. This observation was made before by Cook in his more theoretical approach to the semantics of inheritance [6].

The conformance of *bufferedport_t* to *port_t* is not necessary for the definitions of the classes *bufferedport* and *port*. Thus, this conformance is not necessary for all inheritance relationships and inheritance must not always cause type-conformance. Only the conformance between *selfbufferedport_t* and *selfport_t* is required.

4 Conclusion

We introduced a concept to model inheritance by parametrical bindings. This creates a typed interface for inheritance relationships. The types of objects can be used for the typing of inheritance, but different conformance relationships must be defined. The typing allows the definition of classes which can replace other classes used as base classes in applications. A new class needs to be type-conforming due to inheritance. Thus, maintenance of library classes becomes more easy, because application programs are not affected by type errors when library classes are changed.

5 References

[1] M.A. Ellis, B. Stroustrup: *The annotated C++ reference manual – ANSI base document*; Addison-Wesley, Reading, Mass., USA; 1990

[2] B. Meyer: *Eiffel: the language*; Prentice Hall, New York, NY; 1992

[3] L. Cardelli: “A semantics of multiple inheritance”; In: *Semantics of Data Types*; G. Kahn, D. B. McQueen, G. Plotkin [Eds.]; Lecture Notes on Comp. Sci. 173; Springer, 1984 – pp. 51-68

[4] W. R. Cook, W. L. Hill, P. S. Canning: “Inheritance is not subtyping”; *Conf. record of the 17th Symp. on Princ. of Progr. Lang.* – POPL, (San Francisco, Cal., Jan. 17-19, 1990); 1990 – pp.125-135

[5] F. J. Hauck: “Inheritance modeled with explicit bindings: an approach to typed inheritance”; *Proc. of the Conf. on Object-Oriented Progr. Sys., Lang., and Appl.* – OOPSLA, (Washington, D.C., Sep. 26-Oct. 1, 1993); SIGPLAN Notices 28(10) – to appear

[6] W. R. Cook: *A denotational semantics of inheritance*; PhD Thesis, Brown Univerisity, 1989