A Systematic Approach to Multiple Inheritance Implementation

J. Templ

Institut für Computersysteme, ETH Zürich, CH-8092 Zürich, Switzerland

templ@inf.ethz.ch

ABSTRACT

Multiple inheritance is commonly believed to increase the expressive power of a programming language, but to slow down efficiency of message sends even in case of single inheritance. This paper shows that none of these arguments hold. First, a technique to express the effect of multiple inheritance in terms of single inheritance is described. Based on this technique an efficient implementation of multiple inheritance is constructed. The resulting implementation does not slow down single inheritance and, in case of multiple inheritance, is more efficient than widely used implementations.

KEY WORDS single multiple inheritance emulation twin objects implementation efficiency

Introduction

The purpose of this paper is neither to defend multiple inheritance nor to attack it but to correct a pro and a contra argument. The pro argument is that multiple inheritance increases the expressive power of a language, the contra argument is that multiple inheritance slows down execution even in case of single inheritance. Both arguments are false as will be shown below. The novel aspect of this paper is the observation that these arguments are closely connected. By finding a way to express multiple inheritance in terms of single inheritance, we can *construct* an efficient implementation. This approach, viz. to start with an emulation and to construct the implemenation systematically, simplifies the process of teaching and understanding multiple inheritance implementation. It also leads, in our oppinion, to a better understanding of what multiple inheritance is all about. Although, after some literature study, it turned out that the derived implementation technique has been published before, [He89, LK89], it seems to be almost unknown in the compiler writers community. It is especially surprising that a less efficient technique is still used for C++ [Str87].

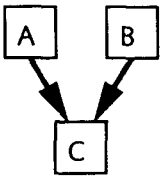
In order to avoid the discussion of orthogonal problems we reduce the scope of this paper to independent multiple inheritance, i.e. the case where the super classes of a class form a tree rather than a lattice. In other words, we don't talk about repeated inheritance or virtual base classes. We also reduce our considerations to statically typed object-oriented programming languages, i.e languages like

Simula–67, C++, or Oberon–2.

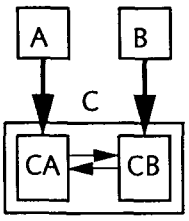
For an introduction to the problems of implementing multiple inheritance please refer to [Kro85].

Twin objects

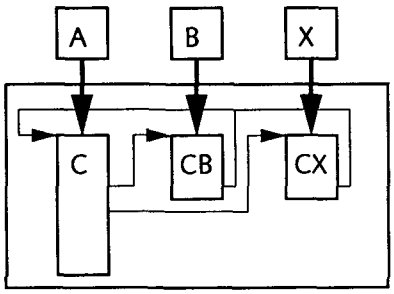
We show a simple technique to express the effect of multiple inheritance by relying on single inheritance only. The technique is called *twin objects* [Moe93], because it essentially consists of the idea to represent instances of a class that inherits from let's say two super classes by a set of two closely related objects, called twins. The name *twin* expresses the fact that these objects are always generated together. If inheriting from more than two base classes, there may also be triplets or in general *n*–tuples of related subobjects. Suppose we want to implement a class *C* that inherits from two classes *A* and *B*.



It must be possible to assign objects of class *C* to objects of both class *A* and class *B*. It must also be possible to override inherited methods from both *A* and *B* in *C* and to access all inherited instance variables and methods. When using single inheritance to express this situation, this can be done by representing class *C* by two subclasses, one derived from *A* and the other derived from *B*. Thus class *C* does not exist as such but is represented by the pair of classes (*CA*, *CB*). As instances of class *C* must form a logical unit, we need a way to refer from one of the twins to the other. This can simply be done by introducing in classes *CA* and *CB* an instance variable (a pointer) that refers to the other twin.



This arrangement allows us to express assignment, type test, selection of inherited components, and method overriding. If we are going to introduce additional instance variables and methods in class *C*, we can define an additional class for C specific extensions, or even simpler we can put the extensions in one of the twin classes, e.g. *CA* and call it *C*. It is also possible to extend the above arrangement to more than two base classes as shown in the figure below.



The following pseudo code example shows the most important operations for a class *C* that inherits methods *P* and *Q* from classes *A* and *B*.

```
CLASS A; ...
   PROCEDURE (self: A) P; ...

CLASS B; ...
   PROCEDURE (self: B) Q; ...

CLASS C(A);
   b: CB;

CLASS CB(B);
   c: C;

VAR a: A; b: B; c: C; cb: CB;

Allocation of C objects:
   NEW(c); NEW(cb); c.b := cb; cb.c := c;

Assignment to variable of less specific type (c may be NIL).
   a := c;
   IF c # NIL THEN b := c.b ELSE b := NIL END;

Type test and assignment to variable of more specific type. The type test (v IS T) returns TRUE iff v (where v
# NIL) is at least of dynamic type T. We use the type guard v(T) to express that v must be at least of
dynamic type T and therefore can be regarded as if it had the static type T.
   IF a IS C THEN c := a(C) END ;
   IF b IS CB THEN c := b(CB).c END ;

Overriding method P:
   PROCEDURE (self: C) P; ...

Overriding method Q:
   PROCEDURE (self: CB) Q; ...

Adding method R to C:
   PROCEDURE (self: C) R; ...

Message sends:
   c.P; c.b.Q; c.R;
```

By applying this schema it is possible to express multiple inheritance in terms of single inheritance. Of course, it requires a certain programming effort, viz. one extra subclass for each base class. In case of inheriting from a twin class, the derived class becomes a twin class, too. Allocation of objects becomes more complicated as it involves allocation of multiple subobjects and installation of references between subobjects. Dealing with such objects also requires pointer dereferencing in various places.

Nevertheless, this schema shows that multiple inheritance is purely syntactic sugar for expressing situations where objects have a one to one relationship and refer to each other. It is a matter of taste whether to introduce additional syntax into a programming language or not. Certainly, it would not

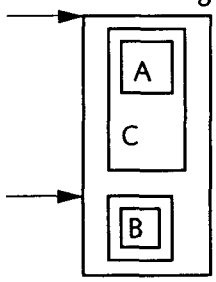
increase the expressive power of a language. It may, however, increase the efficiency of an implementation as shown in the following chapters. It will also lead to a more compact notation because the auxiliary classes need not be defined explicitly but can be generated by the compiler.

Optimizations

Assuming some language support for multiple inheritance, twin objects can be implemented more efficiently. We show the optimization steps that lead to an efficient implementation of multiple inheritance without affecting the implementation (and thereby the efficiency) of single inheritance. We suppose that the reader is familiar with implementation techniques for single inheritance (method lookup tables) in statically typed languages. For an introduction see [Str87].
We try to develop our implementation by a sequence of steps.

- Group all subobjects (twins) of a compound object into one storage block in order to speed up the allocation of objects. This implies that there may be references pointing inside a storage block.
- The pointers connecting the subobjects can be replaced by offsets relative to the beginning of the subobject (positive and negative values).
- If all instances of a given class (and subclasses thereof) use the same allocation order of subobjects, the offsets become compile time constants that need not be stored within the objects.
- Following pointers to subobjects has to be be replaced by constant offset addition.
- The method tables of all subclasses introduced to express multiple inheritance can be stored contiguously in one block.

The following picture shows a possible layout of subobjects for the previously mentioned example. Note that subclasses *C* and *CB* still exist but there are no pointers that refer between subobjects. The subclass *CB* is only used for overriding of inherited *B* methods and for type tests. The method tables are not presented here, because they look exactly like the method tables for single inheritance implementation.



All the details of when to increment a pointer, when to check for NIL, how to perform type tests and so on follow naturally from the emulation technique.

Implementation Note 1: On most three address architectures (e.g. RISC machines) constant addition can be done together with register move in one machine cycle. Such register moves occur e.g. in reference assignment or in parameter passing (including the receiver parameter) if both source and destination variables are allocated within registers. Thus constant offset addition does not need extra machine cycles in many cases.

Implementation Note 2: If storage has to be reclaimed automatically, care has to be taken to handle twin objects correctly. For a mark and scan collector, a simple implementation can be found, that does not introduce any garbage collection overhead for single inheritance. It is sufficient to retain the pointers referring between subobjects to make the mark phase work. After marking, either all subobjects of a compound object are marked or they are all unmarked. For the scan phase we have the additional

requirement, that a compound object must look like a sequence of independently allocated subobjects. This leads to unmarking or freeing of a compound object by unmarking or freeing of its subobjects respectively. It requires, however, that the compiler observes system specific constraints for storage allocation (e.g. minimal block size, alignment restrictions) when calculating the layout of compound objects.

Overriding conflicting methods

Inheriting a method from more than one base class constitutes a conflict that must be handled by the programming language semantics. If the language allows (or requires, e.g. C++) to override such methods, it must be done by overriding each of the conflicting methods individually. In order to avoid code duplication, it is reasonable to use forwarding methods as sketched in the example below.

```
PROCEDURE (self: A) P; ...
PROCEDURE (self: B) P; ...
PROCEDURE (self: C) P; ... overriding method P
PROCEDURE (self: CB) P; self.c.P END P;
```

Class *C* inherits method P from two base classes *A* and *B*. In order two resolve the ambiguity, P has to be overridden by the same method in both class *C* and *CB*. A message *P* sent to an object of dynamic type *CB* is forwarded to *self.c* which activates the overridden method in class *C* and avoids code duplication. The involved runtime overhead seems to be justified assuming this case to be rather exceptional.

Comparison to C++ Implementation

Probably the most widely known implementation of multiple inheritance in a statically typed object oriented language is the C++ implementation described in [Str87]. Actually, this implementation is derived from an earlier proposal made for Simula [Kro85]. The C++ implementation founded the common misbelieve that an efficient implementation of multiple inheritance always slows down the implementation of single inheritance. [Str87], however, does not claim that it describes the most efficient implementation (nor does it claim that multiple inheritance is proven to be an important structuring tool).

The main invariant in the C++ implementation is that *self* (the variable denoting the receiver of a message) always points to an object of the class the method belongs to. Referring to the above example, methods *P* and *Q* in class *C* (inheriting from *A* and *B*) are both compiled as if *self* points to the beginning of a compound *C* object. This is in contrast to our approach which assumes that the receiver is an object of the class that introduced the message or a subclass thereof derived by single inheritance. Objects in C++ are stored in one contiguous block very much like in our approach. In order to establish the main invariant in C++ message sends, the compiler has to generate code that adjusts the *self* parameter according to the static and dynamic type of the receiver (similar to the forwarding methods in the previous chapter). In many cases, most notably in the case of single inheritance, this means the addition of a variable containing value zero. The advantage of this technique is when overriding a method inherited from more than one base classes, because only one method needs to be compiled. This, however, is considered an optimization of a rather exceptional case, which does not justify any overhead for all remaining cases.

Conclusions

It has been shown that independent multiple inheritance can be expressed by multiple application of single inheritance. Support for multiple inheritance in a programming language does not increase the expressive power of the language, but simplifies the implementation of structures like twin objects. It also allows a more efficient implementation which is based on (but does not affect) the implementation of single inheritance. The efficiency differences between an emulation, the C++ implementation, and our approach are, however, almost not noticeable. Thus the advantage of multiple inheritance support in a programming language is purely syntactical.

Acknowledgements

The technique to express multiple inheritance by multiple application of single inheritance has been developed in close cooperation with R. Griesemer and H. Mössenböck. I would also like to thank C. Szyperski for encouraging me to write this paper.

Bibliography

[Str87] B. Stroustrup,
Multiple Inheritance for C++,
Proceedings EUUG spring conference, May 87.

[Kro85] S. Krogdahl,
Multiple Inheritance in Simula–like Languages,
BIT 25 (1985), pp 318–326.

[LK89] K.H. Lee, D. Kafura,
A Fast and Efficient Method Dispatching for Statically Typed Multiple Inheritance Object–Oriented Languages,
TR 89–40, Virginia State University, Dept. of Computer Science.

[He89] B. Heeb,
Multiple Inheritance for an Object–Oriented Extension of Modula–2
Proceedings of the 1st International Modula–2 Conference, 1989, Bled, Yugoslavia.

[Moe93] H. Mössenböck
Object–Oriented Programming in Oberon–2
Springer Verlag, to appear in 1993.