Position Paper for ECOOP'92 Workshop on Multiple Inheritance and Multiple Subtyping

Multiple Inheritance and Type System Design

Andrew P. Black
black@crl.dec.com
Digital Equipment Corporation
Cambridge Research Laboratory

It is now widely accepted that inheritance is not subtyping [5], and that "inheritance is a relationship between implementations, while conformity is a relationship between interfaces" [1]. But like most maxims, these two are something of an oversimplification.

Inheritance: Mechanism or Relation?

As realized in Simula and Smalltalk, inheritance is a *mechanism* for generating new pieces of program from old pieces of program. As such it is of tremendous value to those who write programs (as opposed to those who write papers). In contrast, subtyping is a relation between objects; whether or not one object's type conforms to another object's type depends on the interface of those objects but not on the code that created them. In this sense it is obvious that inheritance is not subtyping. But in a wider sense, inheritance gives rise to a relation that *is* similar to subtyping. Following Bruce[2] we will call this relation **inh**; figure 1 shows how **inh** is related to inheritance.

Suppose that we have a piece of program $\mathcal{A}$ from which we construct through the *mechanism* of inheritance another piece of program $\mathcal{AB}$. As Cook [4] has shown, $\mathcal{A}$ and $\mathcal{AB}$ must be thought of not as classes, but as class *generators*, i.e., as functions from classes to classes. In programming language terms, this means that $\mathcal{A}$ and $\mathcal{AB}$

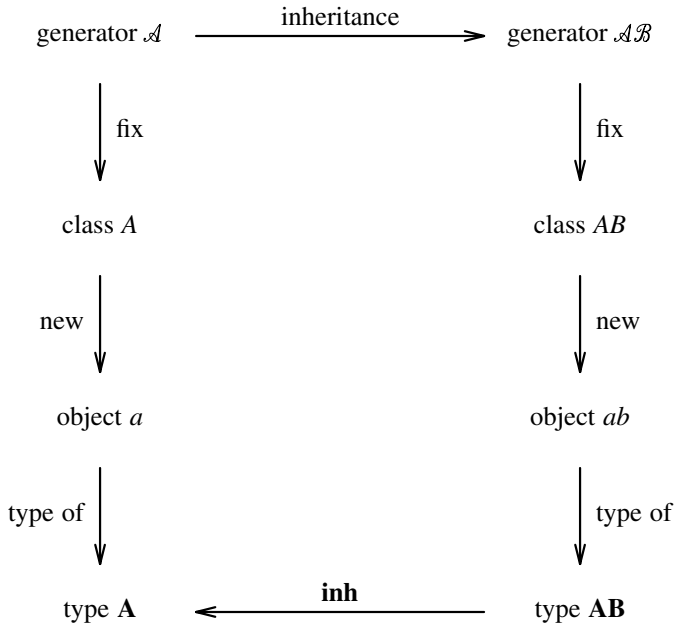

Figure 1. Generator $\mathcal{AB}$ is created from generator $\mathcal{A}$ by the mechanism of inheritance. Taking the fixpoint of $\mathcal{A}$ closes it up to give the class *A*. Applying new to *A* generates an object *a*, which can be characterized by the type **A**. If the corresponding operations create object *ab* with type *AB*, then *AB* **inh** *A*.

contain local names (such as *myclass*) that must be given a meaning before any objects can be created, but which must remain uninterpreted if inheritance is to take place.

Formally, classes are made from generators by taking *fixpoints*; informally we can think of an operation *fix* that takes a generator and creates from it the corresponding class by fixing the meaning of the name *myclass* to be the very class just created.

Given the classes *A* and *AB*, there will be some mechanism (called *new* in figure 1) by which they can be used to create objects *a* and *ab*, which can be characterized by the types **A** and **AB**. The relationship between these types is shown by the solid horizontal line labeled **inh**. The exact relation that **inh** denotes depends on two things: the characteristics of objects that are captured by the type system, and the changes that are allowed by the inheritance mechanism. We will consider each in turn.

Generalized Type Systems

We have long argued that the rôle of types in object-oriented languages is to prevent "message not understood" errors [1]. To accomplish this, types must describe the operations that objects *do* understand, and the arguments and results of those operations, and so on recursively. Types that serve in this rôle might be called "interface types". An interface type system is then a formal tool for reasoning about object interfaces.

The view that types define implementation data layouts goes back at least to the early days of Fortran. It is just as much mistake to reject this kind of type as overly pragmatic as it is to reject formal specifications just because they need to be interpreted by people. Someone, usually the compiler writer, must reason about data layouts, and there is no reason not to give him or her a formal system as an aid to such reasoning. Types that have this rôle might be called "data layout types".

Interface types and data layout types are just two points in a large space of "generalized types". "Specification types" that formally define the semantics of the operations are a third point. There are as many different kinds of type system as there are sorts of programs properties about which one might wish to reason. Nevertheless, for the purposes of this paper, these three will be sufficient.

Modification by Inheritance

Different programming languages allow different sorts of modifications to be made when inheriting existing code. Choosing the "right" set of modifications is a tricky design problem. On the one hand, if there are many restrictions on the kind of modification that is possible, then inheritance looses much of its value. On the other hand, if there are no restrictions at all on inheritance, then a program created through inheritance might be completely unrelated to its parent. We will use the three kinds of types described above to look at various kinds of modifications.

Considering first the changes that can be measured by interface types, the following appear to be reasonable:

- reinterpreting the meaning of the self-reference *mytype*;
- adding a new operation; and
- subtyping the signature of an operation.

Adding a new method and reinterpreting self-references are in some sense the "essence" of inheritance, and it is exactly this combination of modifications that causes the **inh** relation induced by inheritance to differ from the subtyping relation in an interface type system[5]. Allowing the subtyping of an operation signature, but disallowing more general changes, does not help to make **inh** the same as subtyping; these relations would differ even if operation signatures could not be modified at all. The motivation for the restriction on the changes that can be made to the signature of an operation ω are that more general changes might invalidate calls of ω from within the body of another operation. Given the subtyping restriction, Bruce [2] shows

$$\forall p.\ (\text{generator}\ \mathcal{AB}(p) \leq \text{generator}\ \mathcal{A}) \ \Rightarrow\ (\text{type}\ \mathbf{AB}\ \mathbf{inh}\ \text{type}\ \mathbf{A})$$

In words: if, after the self-references in the generators have been replaced by the same arbitrary parameter, the generators are related by subtyping, then the types of the objects that they manufacture will be related by **inh**. Note that this definition allows **AB inh A** if objects with types **A** and **AB** *could have been* created from generators $\mathcal{A}$ and $\mathcal{AB}$; there is no requirement that $\mathcal{A}$ or *AB* even exist.

There are several ways in which **inh** and subtyping can be made to coincide. One is to prohibit the reinterpretation of self-references; Modula-3's inheritance mechanism does just this: the **inh** and subtyping relations for Modula-3 are the same[3, 6]. A better alternative is to allow self-references in *positive* (result or post-condition) positions to be reinterpreted, but to prohibit self-references in *negative* (argument or pre-condition) positions. We have previously argued that the interface types of arguments ought to be specified fairly tightly[7]; for example, the *distanceFrom* operation on a point should not require an argument of type *Point*, but rather an argument of type *XY*, where the only operations that are permitted on an *XY* are *x* and *y*, the operations that extract the *x* and *y* coordinates. This is exactly what is required to determine distance. The same reasoning tells us that the types of arguments should not usually be described by a self-reference, but instead by a type that accurately describes the operations that are actually used in the operation body.

Let us now consider the sort of modifications permitted by inheritance using the viewpoint of data layout types. The usual criteria are that new fields can be added to the data layout of the target, but the sizes and offsets of the existing fields must not be changed. This has the consequence that any assertion of the form "field *x* is at offset *n*" that is true of $\mathcal{A}$ will also be true of $\mathcal{B}$.

In a strongly encapsulated system, code cannot depend on the data layout of any object other than *self*, but it is still necessary to propagate data layout types of arguments and results. For example, an operation that returns the nearer of two argument points *p* (of type *P*) and *q* (of type *Q*) will have a data layout type for its result that is the disjunction of *P* and *Q*; if *P* asserts that the *x* field is at offset 7, and *Q* asserts that the value of the *x* field can be obtained by calling the procedure stored at offset 324, the disjunction may well not have enough information to ensure that the compiler can generate efficient code to access the result directly.

Because of this problem, languages that use types as a mechanism for communicating data layout information as well as interface information usually insist that each interface have exactly one implementation. This removes one of the great benefits of object-orientation: the ability for differing implementations of the same interface to be treated identically.

Multiple Inheritance

The above discussion has been about inheritance in the abstract; it is as true for multiple inheritance as for single inheritance. If a generator $\mathcal{ABC}$ inherits from $\mathcal{A}$ and $\mathcal{B}$, then we expect type **ABC inh** type **A** and type **ABC inh** type **B**. Such expressions can be simplified if there are conjunction and disjunction operations on the various kinds of types.

For example, when dealing with interface types, the conjunction of a type **Z** that has operations ω and ψ and an type **Y** that has operations ψ and χ is the type **Y**+**Z** with operations χ, ψ and ω. Although some information is lost in working with such upper bounds, it the loss is usually not significant. If the interface type system is constructed with care, we can ensure that all the conjunctions and disjunctions exist; the interface types then form a lattice. (This requires, for example, that the same object can possess both an operation + with zero arguments and an operation + with one argument. There are various ways to achieve this.)

However, when working with data layout types, it is hard to see how the types can be embedded in a lattice in such a way that much useful information will be retained. What is the conjunction of *P* and *Q* above? They are contradictory, so it seems that the answer must be **false**. This does not provide much information for the compiler.

Conclusion

In the presence of multiple inheritance, we are thus led to the conclusion that the interface and data layout type systems should be distinct. Even the most obdurate language designer, one who has sacrificed object autonomy by refusing to separate these type systems in a language with single inheritance, can hardly be unmoved when faced with the task of incorporating multiple inheritance. Multiple inheritance is therefor to be praised, not only because of the power that it offers to programmers, but because of the salutary lesson that it provides for language designers.

References

[1] Black, A. P., Hutchinson, N., Jul, E., Levy, H. M. and Carter, L. "Distribution and Abstract Types in Emerald". *IEEE Trans. on Software Eng.* **SE-13**, *1* (January 1987), pp.65-76.

[2] Bruce, K. "A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Sematics". Tech. Rep. CS-92-01, Williams College, Williamstown, MA, January 1992.

[3] Cardelli, L., Donahue, J., Jordan, M., Kalsow, B. and Nelson, G. "The Modula-3 Type System". *Conf. Rec. 16th ACM Symp. on Prin. of Prog. Lang.*, January 1989, pp.202-212.

[4] Cook, W. R. "A Denotational Semantics of Inheritance". CS-89-33, Dept. of Computer Science, Brown University, Providence, Rhode Island, May 1989.

[5] Cook, W. R., Hill, W. L. and Canning, P. S. "Inheritance is Not Subtyping". *Conf. Rec. 17th ACM Symp. on Prin. of Prog. Lang.*, January 1990, pp.125-135.

[6] G. Nelson, ed., *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[7] Raj, R. K., Tempero, E. D., Levy, H. M., Black, A. P., Hutchinson, N. C. and Jul, E. "Emerald: A General Purpose Programming Language". *Software — Practice & Experience* **21**, *1* (January 1991), pp.91-118.