The Structural Semantics of Inheritance

Kenneth Baclawski

College of Computer Science
Northeastern University
Boston, Massachusetts 02115

Abstract

A large variety of different concepts have been introduced to express the notion of inheritance. Single inheritance, multiple inheritance, multiple independent inheritance and delegation are all various forms of the concept of inheritance. In this paper, we give a structural framework for the semantics of these forms of inheritance. This framework accounts for the structural and for some of the behavioral characteristics of the many forms of inheritance.

1. The Many Forms of Inheritance

Most artificial intelligence systems as well as many programming languages and database management systems support techniques whereby one object or type can inherit from another object or type in some fashion, and a great diversity of distinct inheritance concepts have been proposed and implemented. The purpose of this article is to present a framework that captures some of this diversity so that the concepts can be compared. Our point of view is structural, but some of the behavioral aspects are also considered. For an excellent treatment of the behavioral aspects of inheritance in programming languages see [WE89]. For inheritance in artificial intelligence see [4]. Most of the examples will use C++ [6] because this is the most common programming language supported by commercial object-oriented database management systems.

We begin with a list of some of the variations on the general theme of inheritance. In this list, the concepts are merely introduced. We will discuss each concept in more detail in later sections.

TAXONOMY. The *taxonomy* or *classification hierarchy* or *generalization hierarchy* has been used in biology and other sciences for centuries to classify observations and phenomena. More recently it has been introduced into computer science where it has been used for the same kinds of tasks, but it has also been generalized in many ways.

SINGLE INHERITANCE. This is called *simple public derivation* in C++. This technique allows a type to inherit attributes from one other type, and in addition, supports additional features such as *substitutability* and *late binding*.

MULTIPLE INHERITANCE This is also called *virtual public derivation* in C++. This technique allows a type to inherit attributes from several other types, while still supporting substitutability and late binding.

MULTIPLE INDEPENDENT INHERITANCE This is the default form of public derivation in C++. It differs from multiple inheritance in a number of ways, yet still supports substitutability and late binding.

DELEGATION This concept is also known as *prototype inheritance* or *extension inheritance*. It allows an object of one type to be used as a *prototype* for an object of another type, which *extends* the prototype. The extension can *override* attributes of the prototype. The extension object could also have the same type as the prototype, in which case it is a *version* of the prototype. The technique has been proposed as a way to support versions of objects.

PRIVATE INHERITANCE. This is also called *private derivation* in C++. This mechanism is syntactically similar to multiple inheritance, but it does not have substitutability nor does it allow late binding.

We now introduce a data model which is then used to express all of the forms of inheritance mentioned above.

2. The MU Model

The `mu` model is one of the components of the `nu&` semantic object-oriented data modeling tool [1], [2], [3]. This data modeling tool is intended to be a vehicle for both research and education on such tools. It is concerned with studying ways to enhance the modeling power of a transaction management system while maintaining its high-performance characteristics. The name is a reference to Northeastern University where most of the work is being carried out.

The `mu` model is an object-oriented semantic data model. The introduction to given in this section leaves out some details for the sake of a smoother exposition. A rigorous introduction to the `mu` model is given in the Appendix (section). For a more thorough treatment see [3].

A *database* in the `mu` model is defined to be a set D of triples (o, a, p), where o and p are *objects* and a is an *attribute* or *component*. We will use the graphic notation $o \xrightarrow{a} p$ to mean that (o, a, p) is in the database D, and we will say that the object o has *value* p for the attribute a, or more succinctly, that o has a-value p. Notice that attributes are *multivalued*: an object o can have many a-values. The universe of all possible objects is denoted $\mathbf{O}$, and the universe of all possible attributes is denoted $\mathbf{A}$.

Using the terminology of object-oriented programming, the concept of an attribute includes both instance variables (called data members in C++) and methods (called function members in C++). If $o \xrightarrow{a} p$ and a represents an instance variable, then p is the value of the instance variable a in the object o. If a represents a method, then p is the return value of a message sent to the object o using the method a. The `mu` model can represent message parameters and binding times to some extent, but these behavioral aspects of object-oriented programming will not be discussed in this paper.

Most of the concepts introduced in section assume that objects are somehow organized into collections or *types*. As this relationship is so basic, we distinguish it from all other attributes and gave it special features in the `mu` model. This attribute is called the *instance-of* relationship, and it will be denoted by `instance_of` or simply by $a_1 \in \mathbf{A}$. An object o is an instance of a type t in the database D when $o \xrightarrow{a_1} t$.

The `mu` model does not distinguish "object" from "type," so that every object is, in principle, also a type. However, to avoid logical inconsistencies and to have a satisfactory concept of a "schema" from database management and the analogous concept of "compile-time" from programming languages, types and objects are separated from one another. Accordingly, in the `mu` model it is assumed that there is a function called the *level function*,

$$\texttt{level} : \mathbf{O} \longrightarrow \{0, 1, 2, \ldots\}$$

that stratifies the universe of all objects into "ordinary objects" (i.e., those for which `level` has value 0), "types" (i.e., those for which `level` has value 1), "meta-types" (i.e., those for which `level` has value 2), and so on. The attribute a_1 relates objects on adjacent levels:

$$\text{if } (o, a_1, p) \in D \text{ then } \texttt{level}(p) = \texttt{level}(o) + 1.$$

All other attributes are constrained to relate objects on the same level. In this paper, we only consider levels 0 and 1, and we refer to them as the "object level" and "type level," respectively.

To express the concept of a schema or type specification, instances of a type must be constrained by the attributes of the type. For example, suppose that `person` and `string` are two types. We would like to specify that an instance of `person` has a `name` which must be of type `string`. This is done by inserting the triple (`person`, `name`, `string`) into the database D on the type level. We call this a *type constraint*: it *constrains* any `name` value of an instance of `person` to be an instance of type `string`. Type constraints are important enough to make them an axiom of the `mu` model called the *type constraint axiom*: if (t, a, u) is in the database and o is an instance of t, then every a-value of o must be an instance of u.

The type constraint axiom does not fully capture the concept of a type specification, since it allows instances of a type to have attributes other than those attached to the type. Moreover such "extra" attributes are unconstrained. There are situations where such attributes are useful, for example, when types are considered instances of a meta-type. However, for ordinary objects one generally assumes the *type specification axiom*: if (o, a, p) is in the database, then there are types t and u such that o is an instance of t, p is an instance of u and (t, a, u) is also in the database.

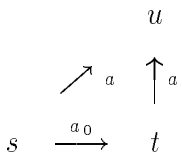
The type constraint and the type specification axioms apply only to "ordinary" attributes. This means that they do not apply to the `instance_of` attribute, nor to the `is_a` attribute to be introduced below.

The concept of inheritance is represented with a distinguished attribute that has special features in the `mu` model. This attribute is called `is_a` or $a_0 \in \mathbf{A}$. When (s, a_0, t) is in the database, we say that s is *derived* from t and that t is a *base* or *base component* of s. When s and t are types, we say that s is a *subtype* of t and that t is a *supertype* of s.

We assume, as an axiom, that the `is_a` attribute is *acyclic*. In other words, there are no `is_a` cycles. Some models allow such cycles. The objects in an `is_a` cycle are *synonymous* (i.e., equivalent to one another) with respect to the data model. While there are uses for synonymous objects, it is both simpler and more flexible to allow objects to have several "names" than to allow objects to be synonymous, where the name is one of the attributes of the object. For this reason, we require a_0 to be an acyclic attribute, and therefore do not allow objects to be synonymous.

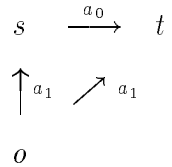
There is one more axiom for the `mu` model, the *inheritance axiom*. Unlike the others, there are three possibilities for this axiom. For each choice of this axiom, we obtain a slightly different data model. Note that these are not different forms of inheritance, but rather different semantic data models within which one can try to express the various forms of inheritance.

Variation A of the inheritance axiom will also be called *attribute inheritance*. This version of the axiom postulates that subtypes acquire or *inherit* the attributes of all supertypes. More precisely, if $s \xrightarrow{a_0} t$ and $t \xrightarrow{a} u$, then $s \xrightarrow{a} u$, or graphically:



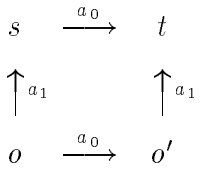
Although one often sees inheritance defined this way, at least informally, it is awkward to express the concepts of substitutability and late binding using attribute inheritance, and almost impossible to express attribute overriding. So in practice, one of the other two variations is actually used.

The second variation is called variation I or *instance inheritance*. This postulates that instances of a subtype are also instances of any supertype. More precisely, if $o \xrightarrow{a_1} s$ and $s \xrightarrow{a_0} t$, then $o \xrightarrow{a_1} t$, or graphically:



If one thinks of types as collections of objects, then this axiom is set-theoretic: supertypes contain subtypes.

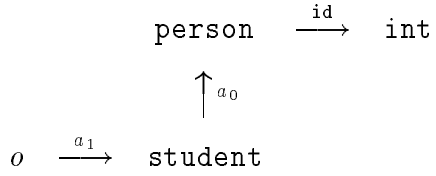
The third variation, called variation O or *object inheritance*, seems quite different from the other two. In this variation, the inheritance graph on the type level is reproduced for each object instantiated on the object level. More precisely, if o is an instance of s and s is a subtype of t, then there is a unique object o' such that o' is an instance of t and o is derived from o'. Graphically it looks like this:



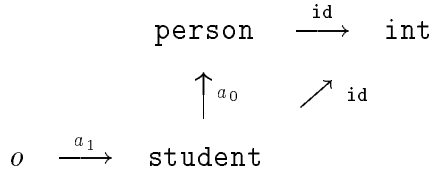
We call o' the *base component* for o of type t. One can think of object inheritance as an implementation of instance inheritance, but it is more than that. As we will see there are forms of inheritance that can be expressed in terms of object inheritance but that cannot easily be expressed using attribute or instance inheritance.

Before comparing the three variations, we say a few words about the words (i.e., the terminology). The term "inheritance" intuitively suggests variation A. On the other hand, the term "is a" intuitively suggests variation I. Finally, variation O is not as intuitive as the other two, but the term "delegation" seems to fit reasonably well.

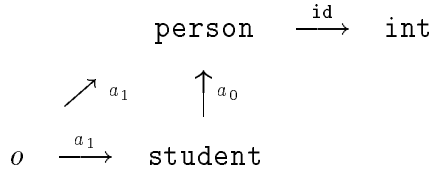
The three variations differ in their expressive power. Consider this simple situation: we have a type `person`, a subtype `student` and an object o of type `student`. Suppose that one of the attributes of `person` is `id` of type `int`. In terms of diagrams:



In variation A, o has attribute `id` of type `int` by virtue of the fact that `student` has this attribute:

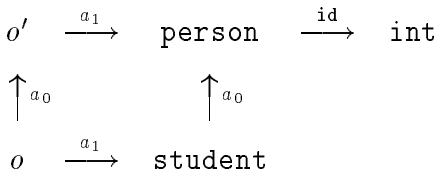


In variation I, o has attribute `id` of type `int` because o is also an instance of type `person`:



In this variation, `student` could either have the attribute `id` or not, thereby allowing a subtype to "override" an attribute in a supertype, something that cannot be expressed in variation A. However, overriding can only put an additional type constraint on this attribute (for example, that the `id` must be in a certain range).

Finally, consider variation O. In this case, there is a unique base object o' of type `person`:



Here, o has attribute `id` indirectly through the base object o'. As in variation I, `student` could either have the attribute `id` or not, thereby allowing a subtype to "override" an attribute in a supertype. Now, however, as an attribute of `student`, `id` could have any type. Moreover, o' will still have its own attribute value(s) for `id` which can be uncovered under certain circumstances.

We will henceforth assume that variation O of the inheritance axiom holds unless specified otherwise.

To complete the description of the `mu` model one must distinguish a number of *built-in* objects. For example, one must specify built-in types such as `int`, `string` and so on. The built-in types represent the starting point for building more complex types. The choice of which types should be built-in is an important design decision of any data model, but for the most part these will not be relevant to this article.

Dual to the concept of built-in type is the concept of a built-in constraint. To allow constraints to be manipulated like any data in the database, they are given object status. Like objects in general, complex constraints can be built from more elementary constraints. Some of the built-in constraint objects for the `mu` model will be introduced in the next section.

3. Components

The concept of a component or attribute is a basic structuring technique for virtually every programming language or database management system. It goes by a variety of names, but there is a remarkable agreement about what it should mean. This is in contrast to the concept of inheritance where the same name is used for a great variety of different concepts about which there is considerable disagreement.

As we noted earlier, every attribute in the `mu` model is multivalued by default. It is important to distinguish between attributes that must be single-valued from those that can be multivalued. Such a constraint on an attribute is called a *functionality constraint*. In the `mu` model a functionality constraint is a built-in object on the type level. The most important functionality constraints are the following:

ONE-TO-MANY. An attribute a of a type t is *one-to-many* if no two instances of t have the same a-value. Such an attribute represents a *containment* relationship, since there is no sharing of attribute values among instances of t.

MANY-TO-ONE. An attribute a of a type t is *many-to-one* (or *single-valued*) if every instance of t has at most one a-value. Most attributes are single-valued, and this is the only form of attribute available in many programming languages and database management systems.

ONE-TO-ONE. An attribute is *one-to-one* if it is both one-to-many and many-to-one.

MANDATORY. An attribute a of a type t is *mandatory* if every instance of t is required to have at least one a-value.

The fact that the same attribute can be used for several types leads to the possibility of ambiguity. A *trait* is a pair consisting of an object and one of its attributes (see the Appendix for a rigorous definition). Traits are never ambiguous while attributes might be because attributes can be overridden. In order to deal with this problem, attributes and hence traits are accessed using a naming system. The relationship between traits and their names is many-to-many: each trait has many names and different traits can have the same name.

In general, names of attributes can be complex. On the type level, an attribute name usually consists of an identifier or an identifier together with the name of a type for which it is a component. For example, in C++ the attribute `address` of `person` has two names: "`address`" and "`person::address`." On the object level, the names of traits can be even more complex: in addition to the two already mentioned on the type level, there can be names that specify a sequence of increasingly more derived types. The `mu` model provides mechanisms for specifying the many-to-many relationship between names and traits. The module in the `nu&` system responsible for supporting these features is called the *name server*, and disambiguation is called *name resolution*. This allows the disambiguation mechanism to be tailored to a programming language more easily, since each programming language has its own mechanism for disambiguation. By separating attributes and traits from their names, the `mu` model makes it possible to integrate the model into diverse object-oriented programming languages.

4. Properties of Inheritance

Inheritance, especially when combined with properties like substitutability and late binding to be discussed below, is one of the most important features provided by an object-oriented system. A system must support this concept if it is to be considered object-oriented.

Assume for the moment that we are using variation I of the inheritance axiom. In this case, the `instance_of` attribute will be multivalued. For example, suppose that `student` and `teacher` are subtypes of `person`. If an object is instantiated as a `student`, then it will also be an instance of `person`. While the object is an instance of several types, it is an instance of exactly one *most derived* type, namely `student`. Most object-oriented programming languages would not allow an object to be an instance of both `student` and `teacher` without being an instance of some type derived from both of these. In other words, these programming languages require that objects satisfy what we call the *unique type axiom*: for every object o such that `level`$(o) = 0$, there exists a unique most derived type s, such that $o \xrightarrow{a_1} s$.

The fact that `instance_of` is multivalued in object-oriented systems makes "the type of an object" ambiguous. To clarify this, CLOS [5], uses the term "member of" for what we have been calling "instance of," while the term "instance of" in CLOS is reserved for what we would call "instance of the most derived type." Using CLOS terminology, an object can be a member of many types, but it is an instance of exactly one. This terminology is

appealing, because it fits well with the set-theoretic interpretation embodied in the instance inheritance axiom.

4.1. Substitutability

An important property of any concept of inheritance is *substitutability*: any instance of a subtype can be substituted for (or regarded as) an instance of a supertype. This property is hard to express using variation A. It is trivial for variation I, since every instance of the subtype *is* an instance of the supertype.

By contrast, substitution is a nontrivial operation for variation O, involving the replacement of the original object by one of its base components. The implementation of inheritance in C++, for example, is accomplished by using pointer manipulation. The C++ implementation is an example of "eager" substitution: the original object is replaced by its base component at the time substitution occurs. Some other systems use "lazy" substitution: the replacement is delayed until an attribute is evaluated.

4.2. Late binding

Another important property of inheritance is support for *late binding*. A *late bound function* (or *method*) is a certain kind of function which can "remember" the original object when invoked on a base component obtained by substitution. Here is an example of this concept. Let `student` be a type derived from the base type `person`. Suppose that each type has its own function component called `display` that prints information about an instance. One would like to keep lists of persons, some of whom might be students, and to print information about them using the display function. A variable `x` of type `person` is used to scan through such a list and the `display` function is invoked for each object in the list. Since the variable `x` is of type `person`, it is necessary to regard each instance of `student` in this list as a `person`. In other words, one is using substitutability to construct the list. On the other hand, when `display` is invoked one would like the `student` function to be used when an object is a student and the `person` function to be used when an object is a person. In other words, the result of a substitution must still "remember" the original object.

The mechanism for remembering the original object of a base component will be called *recollection*. Intuitively it undoes substitution, but there are a number of circumstances that can complicate recollection. For example, one can substitute several times in succession to obtain a sequence of base objects while recollection always gives the original object in a single step. In addition, there are forms of inheritance where the unique type axiom does not hold, in which case there is no most derived object to recall. Finally, one can sometimes specify explicit conversions that perform only a partial recollection.

One of the features of the `mu` model is that late binding is split into two independent steps: recollection and name resolution. By modularizing late binding in this manner, the concept is simplified and much easier to implement. Name resolution may involve substitution, but never involves any recollection of an object, while recollection is concerned solely with objects and does not involve any names at all. For a precise definition of recollection and partial recollection, see section . Recollection and partial recollection are multivalued in general. If recollection is not single-valued, then there must be a disambiguation mechanism for recollection to be possible, and similarly for partial recollection.

Late binding allows one to write procedures that exhibit a limited but very useful form of *polymorphism*. A procedure is *polymorphic* if it can act on a variety of types without being rewritten or even recompiled.

As is the case with substitutability, recollection is hard to express using variation A, and it is trivial for variation I. Recollection is also relatively easy to express in variation O, provided that the unique type axiom holds. Even partial recollection is not difficult: think of it as a full recollection followed by a substitution. However, if the unique type axiom does not hold, then recollection and partial recollection may depend on the substitution history of the object.

If "eager" substitution is used and the unique type axiom holds (as is usually the case in C++), then recollection can be accomplished by storing information in the base object. In C++ this information is called the virtual function table pointer. If the unique type axiom does not hold, then recollection requires that the original object still be available after substitution. One solution to this problem is to use "lazy" substitution. However, this approach has an impact on performance relative to the "eager" approach.

The concept of recollection in the `mu` model clarifies some properties of C++ that are otherwise somewhat mysterious. In this language, instantiation of an object is a two-step mechanism. First space for the object and all base objects is allocated. Then the the object and its base objects are "constructed" by calling the appropriate constructor functions. The order in which the constructors are called is not entirely obvious: the constructor for the base components are called before the constructor of the object. However, this requirement is forced by the inheritance axiom: in order for a derived object to have unique base components after being constructed, the base components must be constructed before the object itself. If one of the base components also has its own base components, then the same rule applies to it. Since construction involves many steps, it is possible to invoke the recollection mechanism prior to the completion of construction. The mechanism will result in a well defined object but not, of course, the one that would result if the construction were completed, since that object is the last to be constructed. Virtual functions called during construction of base components will detect this. However, once instantiation is complete, the unique type axiom is satisfied until the object is deleted.

Deletion in C++ is also a two-step mechanism, essentially the reverse of instantiation. First the objects are destructed by calling the destructor functions, normally starting with the most derived object. Then the space used by the object and its base objects is deallocated. However, there is a situation in which some of the destructor functions will not be called. The reason for this surprising situation is that destructors, like other functions, can be either early bound or late bound. If the destructor is late bound, then when an object is deleted, the destructor for the most derived object is called, followed by the destructors for the base objects, as one would expect. However, if a destructor is early bound and a substitution occurred, then the destructors are called as if the base object did not have any derived objects. This seems to contradict the fact that one has control over the extent (i.e., the set of instances) of a type. Yet only a late bound function invokes recollection, so if the destructor is early bound and an object is substituted, then recollection does not occur, and the destructor function for the original object will not be invoked. The lesson from this is

that if one wants full control over the extent of a type in C++, then every base type should declare its destructor to be late bound even if the destructor does nothing.

4.3. Performance independence

Another desirable property of any inheritance mechanism is that if some feature of this mechanism is not used, then the performance of the program should be the same as if the feature was not supported at all. More succinctly, one should not have to "pay" for what one does not use. We will call this property *performance independence*.

For example, in C++, there is an elegant implementation strategy for simple inheritance in which the base object is always implemented as the first component of the object. As a result, the base component has the same address as the object and substitution becomes trivial. Late binding is implemented by storing a pointer in the base component that points to a table of addresses of virtual functions. The table pointed to will depend on the type of the original object. However, if no functions are late bound, then no virtual function table pointer is stored in the object. In other words, if late binding is not used, then there are neither time nor space penalties incurred.

By contrast, if inheritance is implemented using a field containing information about the type of the object (as in Smalltalk), then both space and time penalties are incurred even if one does not use inheritance at all.

Performance independence was an explicit design goal of C++. This might help to explain why destructors are early bound by default, as discussed in section above. Although it is more natural for a destructor to be late bound, this can have an impact on performance.

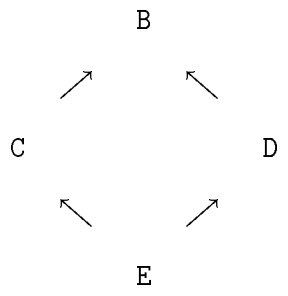
4.4. Transitivity

One property that is seldom discussed in the literature, perhaps because it seems so obvious, is *transitivity*. A database is *transitive* if the `is_a` attribute is transitive, that is, if q is derived from p and r is derived from q then r is derived from p. On the type level, this means that if s is a subtype of t and t is a subtype of u, then s is a subtype of u.

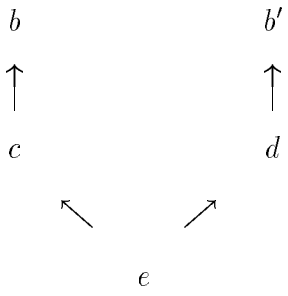
For variations A and I, transitivity is not interesting: replacing the `is_a` graph by its transitive closure adds no additional constraint. By contrast, it does matter for variation O. In fact, transitivity is the sole difference between "virtual" inheritance and "multiple independent" inheritance in C++: in virtual inheritance the `is_a` graph is transitive, while in multiple independent inheritance the graph is nontransitive when two types can be connected by more than one directed path.

Consider the example of four types `B`, `C`, `D` and `E` in C++ such that `C` and `D` are subtypes

of B and such that E is a subtype of both C and D:



Suppose that e is an instance of E. One can substitute e as an instance of C, obtaining an object c, and then substitute c to obtain an instance b of B. However, one can also perform two substitutions via D to obtain an instance b' of B, which need not be the same as b in general (in fact, can never be the same, since the instantiations are independent and C++ does not allow sharing of base components):



If inheritance is assumed to be transitive, then E is also a subtype of B, and the two objects b and b' of B obtained by multiple substitutions must coincide by variation O of the inheritance axiom and the transitivity of a_0.

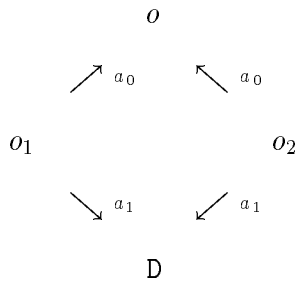
Incidentally, in the example above, all attributes of B are ambiguous in E. To disambiguate such attributes in general, one must specify a sequence of intermediate types. For example, the full name of an attribute of B has the prefix C::B:: when the substitution path goes via C.

In C++ one can mix virtual inheritance with independent inheritance. It is an interesting problem (left for the reader) to characterize those graphs that can be specified in C++ using mixed virtual and independent inheritance.

4.5. Independence of base components

The final property to be considered is whether a base component of an object can be accessed as an object on its own, not just as an object obtained by substitution. A system that has delegation, or more generally any one that allows sharing of base objects, will have this property. For example, suppose that D is a subtype of B. Derived objects o_1 and o_2 *share*

the base object o if



This property has an impact on performance independence: both objects and algorithms must be more complex as a result of the larger collection of possible access paths to an object. In the example above, if o_1 were regarded as an object of type `B` then o_1 must be effectively replaced by o, and similarly for o_2. When the time comes to do recollection, it is not possible to distinguish between o_1 and o_2 using only information in o. As a result, it is necessary to use a more complex implementation of objects. This would inevitably have an impact on performance even when one is not using inheritance, thereby violating the property of performance independence.

Sharing of base components appears to be possible in C++ as an accident of the fact that construction is not an atomic operation. However, sharing of base components is not a supported feature in general, and use of late bound functions during construction is risky.

Sharing of ordinary components is very different from sharing of base components. This form of sharing is easy to support because ordinary attributes do not normally support substitutability and late binding.

5. Classification of Inheritance Mechanisms

One can now give a succinct treatment of the main forms of inheritance, along with a listing of the properties that each form of inheritance supports.

MULTIPLE INHERITANCE. The `is_a` attribute is transitive and one-to-many on the object level. Substitutability and late binding can be supported. Either performance independence or independence of base objects can be supported but not both.

SINGLE INHERITANCE. The `is_a` attribute is one-to-one on the object level. Transitivity, substitutability and late binding can all be supported. Either performance independence or independence of base objects can be supported but not both.

DELEGATION. The `is_a` attribute is many-to-many on the object level. Transitivity, substitutability, late binding and independence of base objects can all be supported. Performance independence cannot be supported.

MULTIPLE INDEPENDENT INHERITANCE. The `is_a` attribute is not transitive and is one-to-many on the object level. Substitutability and late binding can be supported, but only within the parts of the inheritance graph that use single inheritance. Either performance independence or independence of base objects can be supported but not both.

The characterization of (multiple) inheritance mechanisms given above allows one to compare these mechanisms directly with one another. In principle all of the mechanisms

could be implemented in a single system. This is probably not desirable because of the impact on performance in general and because of the violation of performance independence. Nevertheless the framework allows one to compare the modeling power of the various proposed mechanisms as well as to experiment with other inheritance mechanisms.

6. Minor Forms of Inheritance

The remaining forms of inheritance mentioned in section are minor variations on the general theme of inheritance. In this section these concepts are expressed in the `mu` model and compared with the more important forms of inheritance.

6.1. Taxonomy or Classification Hierarchy

A taxonomy is a special case of single inheritance. It requires two additional kinds of constraint: covering constraints and disjointness constraints.

Consider, for example, a database that stores cars and trucks. We define types called `car` and `truck`. We then notice that these two types have some attributes in common. This leads us to make a new type called `vehicle` with these common attributes. This process is called *generalization*. The reverse process in which a more general concept is divided into more specific ones is called *specialization* or *classification*. In either case, these processes lead to a type `vehicle` all of whose instances are either of type `car` or of type `truck`. This is called a *covering constraint*: the types `car` and `truck` *cover* the type `vehicle`. This kind of constraint can be enforced in C++ by declaring the type `vehicle` to be an *abstract data type*, i.e., a type for which there are no (directly accessible) instances.

The second property of a taxonomy is that no two subtypes have a common instance. For example, `car` and `truck` would have no instances in common, if the classification of `vehicle` into `car` and `truck` is a taxonomy. Such a constraint cannot normally be enforced by a programming language: there is nothing to stop someone from deriving a type from both `car` and `truck` and then instantiating objects in this new type. It is curious that taxonomies are often cited as a motivation for inheritance, yet few systems offer the means of constraining a set of types to be a taxonomy.

6.2. Private Inheritance

Private inheritance differs from other forms of inheritance in not supporting substitutability nor late binding. One can argue that the concept is no more than syntactic sugar. From our point of view, it represents a private attribute like any other, but one whose set of names has special properties.

Consider, for example, a class `file` with a private component `filename` of type `string`. In C++ the definition of `file` would look something like this:

```
struct file {
  /* public members */

private:
  string filename;
```

```
  /* other members */
};
```

If it was decided that the `filename` component was so important and used so frequently that it should be given a special status, then one could use private derivation as follows:

```
struct file : private string {
  /* public members */

private:
  /* private members */
};
```

This would permit components of `filename` to be accessed with a simpler syntax. For example, `filename.length()` would become simply `length()` if no component of `file` has this name; otherwise one would have to say `string::length()`. One way of looking at this is that the `filename` component of the first definition becomes an "unnamed" component in the second. In other words, private inheritance can be supported entirely by the name server.

It is interesting to compare private derivation with multiple independent inheritance. The two are not that different, especially when the components of the private base type are "publicized" in the derived type. Consider the example in section . Suppose that `C` and `D` were privately derived from `B` and their attributes publicized. Substitutability and late binding are not allowed in this case, whereas in the case of multiple independent inheritance, one can substitute a `C` for a `B` but not an `E` for a `B`. In other words, private derivation turns off substitutability and late binding entirely while multiple independent derivation turns it off for multiple derivation but not for single derivation.

7. Conclusion

We have presented a single framework for a large variety of forms of inheritance. Having classified and compared all the diverse forms of inheritance, one cannot help but speculate about why the diversity has evolved. One possibility is that all the concepts of inheritance are approximations to the ideal of instance inheritance, differing more because of accidents of the way that inheritance is specified, or the way that inheritance was intended to be used rather than being fundamental to the concept. A more likely explanation is that one would like to have all the properties of inheritance discussed in section , but since they conflict with one another, one is forced to choose which properties will be supported and which properties will be violated.

8. Appendix: Axioms for the `mu` model.

A *database universe* is a quintuple $\mathbf{U} = (\mathbf{O}, \mathbf{A}, \texttt{level}, a_0, a_1)$, where $\mathbf{O}, \mathbf{A}$ are two countably infinite sets and `level` is a function $\texttt{level}\colon \mathbf{O} \longrightarrow \{0, 1, 2, \ldots\}$. We refer to $\mathbf{A}$ as the *set of attributes* and to $\mathbf{O}$ as the *set of objects* of the universe, and we assume

that they are disjoint. The function `level` is the *level function*. We postulate the existence of two distinguished elements of $\mathbf{A}$, denoted by a_0 and a_1. The attribute a_0 is also called `is_a`, while a_1 is also called `instance_of`. Non-distinguished attributes are called *proper* attributes.

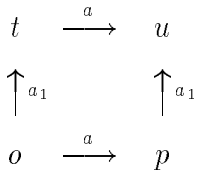
A `mu` *database* (or simply a *database*) of the universe $\mathbf{U}$ is a finite ternary relation $D \subseteq \mathbf{O} \times \mathbf{A} \times \mathbf{O}$, satisfying the following axioms:

Separation of Levels. For every triple $(o, a_1, o') \in D$, $\texttt{level}(o') = \texttt{level}(o) + 1$, and for every triple $(o, a, o') \in D$, where $a \neq a_1$, $\texttt{level}(o) = \texttt{level}(o')$.

Type Constraint. For every proper attribute a, if $(t, a, u) \in D$, $(o, a_1, t) \in D$, and $(o, a, p) \in D$, then $(p, a_1, u) \in D$.

Type Specification. For every proper attribute a, if $(o, a, p) \in D$, and if $\texttt{level}(o) = \texttt{level}(p) = 0$, then there are types t and u such that $(t, a, u) \in D$, $(o, a_1, t) \in D$ and $(p, a_1, u) \in D$.

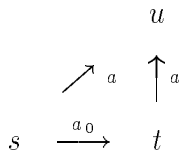
The type constraint and type specification axioms have different assumptions and conclusions, but they have the same diagram:



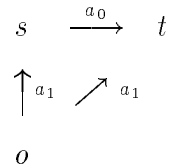
Acyclicity. For any sequence $\{x_1, x_2, \ldots, x_n\}$ of $n > 1$ elements of $\mathbf{O}$, if $(x_1, a_0, x_2) \in D$, $(x_2, a_0, x_3) \in D$, ... and $(x_{n-1}, a_0, x_n) \in D$, then $x_1 \neq x_n$.

Inheritance.

Variation A (Attribute Inheritance). For a proper attribute a, if $(s, a_0, t) \in D$ and $(t, a, u) \in D$, then $(s, a, u) \in D$. Graphically, this looks like this:



Variation I (Instance Inheritance). If $(s, a_0, t) \in D$ and $(o, a_1, s) \in D$, then $(o, a_1, t) \in D$. Graphically, this looks like this:



Variation O (Object Inheritance). If $(o, a_1, s) \in D$ and (s, a_0, t), then there is a unique object o' such that $(o', a_1, t) \in D$ and $(o, a_0, o') \in D$. Graphically it looks like this:

$$\begin{array}{ccc} s & \xrightarrow{a_0} & t \\ \uparrow{\scriptstyle a_1} & & \uparrow{\scriptstyle a_1} \\ o' & \xrightarrow{a_0} & o \end{array}$$

Let a_0^+ be the transitive closure of a_0. In other words, $(x, a_0^+, y) \in D$ if and only if either $(x, a_0, y) \in D$ or there exist objects $z_1, \ldots, z_n$ such that $(x, a_0, z_1) \in D$, $(z_1, a_0, z_2) \in D$, ..., and $(z_n, a_0, y) \in D$. We say that a_0 is *transitive* when $a_0 = a_0^+$. Let a_0^* be the reflexive, transitive closure of a_0. In other words, $(x, a_0^*, y) \in D$ if and only if either $x = y$ or $(x, a_0^+, y) \in D$.

The concept of a *trait* depends on the inheritance axiom as follows:

Variation A. A *trait* is a triple (o, t, a) for which there exists a types s and u such that $(o, a_1, s) \in D$, $(s, a_0^*, t) \in D$ and $(t, a, u) \in D$.

Variation I. A *trait* is a triple (o, t, a) for which there exists a type u such that $(o, a_1, t) \in D$ and $(t, a, u) \in D$.

Variation O. A *trait* is a triple $(o, \{t_0, \ldots, t_n\}, a)$, where $n \geq 0$, for which there exists a type u such that $(o, a_1, t_0) \in D$, $(t_0, a_0, t_1) \in D$, ..., and $(t_n, a, u) \in D$.

The last axiom also depends on the variation used for the inheritance axiom.

Uniquely Typed.

Variation A. For every object o such that $\texttt{level}(o) = 0$, there exists a unique type s such that $(o, a_1, s) \in D$.

Variation I. For every object o such that $\texttt{level}(o) = 0$, there exists a unique type s such that $(o, a_1, s) \in D$ and if $(o, a_1, t) \in D$, then $(s, a_0^*, t) \in D$.

Variation O. For every object o such that $\texttt{level}(o) = 0$,

1. there exists a unique type s such that $(o, a_1, s) \in D$, and
2. there exists a unique object o', (called the *most derived object* of o) such that
   a. $(o', a_0^*, o) \in D$ and
   b. if $(o'', a_0^*, o) \in D$ then $(o', a_0^*, o'') \in D$.

A *recollection* is a multivalued function ρ from **O** to **O** such that

1. if $o' \in \rho(o)$, then $(o', a_0^*, o) \in D$, and
2. if $o' \in \rho(o)$ and $(o'', a_0, o') \in D$, then $o'' \in \rho$.

A *partial recollection* is a multivalued function π from $\mathbf{O} \times \mathbf{O}$ to **O** such that if $o' \in \pi(o, t)$, then o' is an instance of t and $(o', a_0^*, o) \in D$.

References

[1] K. BACLAWSKI, The nu& object-oriented semantic data modeling tool: intermediate report, Northeastern University, College of Computer Science, Technical Report No. NU-CCS-90-18, 1990.

[2] K. BACLAWSKI, T. MARK, R. NEWBY AND R. RAMACHANDRAN, The nu& object-oriented semantic data modeling tool: preliminary report, Northeastern University, College of Computer Science, Technical Report No. NU-CCS-90-17, 1989.

[3] K. BACLAWSKI AND D. SIMOVICI, An algebraic approach to databases with complex objects, *Information Systems*, Pergamon Press, 17(1):33–47, 1992.

[4] R. BRACHMAN, "I lied about the trees" or, defaults and definitions in knowledge representation, *AI Magazine*, pages 80–93, Fall, 1985.

[5] D. MOON, The Common Lisp object-oriented programming language standard, *Object-Oriented Concepts, Databases, and Applications*, Edited by W. Kim and F. Lochovsky, ACM Press (Frontier Series) and Addison-Wesley, Reading, MA, pages 49–78, 1989.

[6] B. STROUSTRUP, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

[7] P. WEGNER, Concepts and paradigms of object-oriented programming, *OOPS Messenger*, 1(1):7–87, 1990.