An Algebraic Semantics of Subobjects

Jonathan G. Rossie, Jr. **Daniel P. Friedman**

Department of Computer Science, Indiana University
215 Lindley Hall, Bloomington, Indiana 47405
{jrossie,dfried}@cs.indiana.edu

Abstract

Existing formalisms of inheritance are not sufficient to model the complexities of the kind of multiple inheritance exemplified in C++. Any satisfactory formalism must model the complicating effects of virtual and non-virtual base classes as well as virtual and non-virtual methods. By abstracting the implementational notion of a subobject and formalizing subobject selection, we develop a formalism to model this combination of features. Not intended as a formal semantics of C++, the resulting model should nevertheless provide an essential level of understanding for language theorists and implementors in their dealings with C++ and related languages.

1 Introduction

The style of multiple inheritance first proposed for Simula by Krogdahl[21] and later developed into the C++ multiple inheritance system by Stroustrup[35, 15] exemplifies a particular kind of inheritance in which the underlying imperative is to maintain the *integrity* of subobjects. *Subobjects* are historically an implementational concept, consisting of the storage for any instance variables introduced by a given class, along with some header information. An instance has, in the simplest case, one subobject for its own class and one for each ancestor class. Although the typical space-optimizing layout allows certain subobjects to share headers, they are still semantically distinct subobjects.

Both authors were partially supported by NSF grant CCR-9302114.

OOPSLA '95 Austin, TX, USA
© 1995 ACM 0-89791-703-0/95/0010...$3.50

Subobject integrity simply means that distinct storage is maintained for every instance variable, even when more than one share the same name. The term is our own, but similar notions appear under different names, including *object integrity*[27] and the *independence principle*[7]. It has been vehemently argued that such a condition is necessary for the adequate separation of implementation and interface[32].

In Krogdahl's model, the effect is that all the method functions inherited along a particular inheritance path can be executed with respect to their own private set of instance variable locations, and thus do not inadvertently interfere with the state-invariants of other classes. Repeated inheritance from the same class along different derivation paths is not permitted, since it is not clear how many subobjects should be created or what should be done about the name collisions if multiple subobjects were created.

Stroustrup extends Krogdahl's model by allowing repeated inheritance, which may cause the instance to have as many subobjects as there are distinct paths. So-called *virtual* classes, which we shall refer to as *shared* base classes, provide a needed modification by allowing a program to express that certain occurrences of repeated inheritance should not result in separate subobjects, but must share a single subobject.

There remains the problem of name collision when the repeated inheritance is not shared, and there is also the problem of incidental sideways name collision between unrelated base classes. Because of subobject integrity, two same-named instance variables inherited along different paths can refer to different storage. Thus instance variable references may be ambiguous. The same problem arises in the case of methods in a slightly different guise: same-named methods from different paths are meant to be executed with respect to their own subobjects; if the naming coincidence forced such methods to be combined, it would not always be clear which subobject to use.

All of these issues contribute to the complexity of the inheritance model. Cargill[6] makes an ample case for the complexity of the system; arguing against the introduction of Stroustrup's multiple inheritance system into C++, he observes (p.71) "Multiple inheritance in C++ is complicated to learn, write and read." He is particularly opposed to shared base classes, which he feels require too much nonlocal information to understand. The complexity is further evidenced by the observable inconsistencies between C++ compilers.

1.1 Models of Inheritance

Most of these issues do not arise in single-inheritance systems, such as Smalltalk[17], nor do they occur in all multiple-inheritance systems. Some multiple-inheritance designs make no effort to maintain the integrity of subobjects. In CLOS[34] and related systems[1, 23, 14], linearization of the inheritance hierarchy results in a collapsing of same-named methods and instance-variables; in some sense, they support only shared base classes.

Formal semantic models of single-inheritance languages [19, 26, 10, 9] are so deeply reliant on there being a single base class that they do not scale up into multiple inheritance. Formal models of multiple inheritance have arisen in the study of type systems for object-oriented languages[3, 4, 5, 25, 8]. Because these models treat objects as records, an object may only associate a single value with each name. Accordingly, such models do not address subobject integrity.

Snyder's model of the C++ object system[33] is similar to ours in that it deals with subobjects on some abstract level (although not quite the same level), but it does not include shared base classes, nor does it model the effects of non-shared repeated inheritance of the same class, which Snyder refers to as a corner case of the language, distinctive to C++ multiple inheritance: (p.10) "... the extra complexity needed to handle this case is not justified."

1.2 Three Questions

Having found no satisfying formal model of inheritance that respects subobject integrity, we proceed to develop our own. Rather than attempting to formalize the entire language, or even the entire object system, we restrict our formalism to the resolution of the three questions that we feel are at the heart of understanding the complexities introduced by subobject integrity.

Question 1 (subobjects) *What is the set of subobjects that comprises an instance of a given class?*

Question 2 (instance variables) *For an instance of a given class and a specific instance-variable name, which subobject will contain the value? (Or will it be ambiguous?)*

Question 3 (methods) *For an instance of a given class and a specific method name, to which subobject will the instance be cast as a result of the call? (Or will it be ambiguous?)*

It is necessary to explain the notion of *casting* in this model. When a class C is instantiated as an instance i, we say that C is the *actual class* of i. As long as i is treated as an instance of C, we say its C subobject is also its *effective subobject* and C is its *effective class*. Using i with a method inherited from an ancestor class A requires that the associated subobject of i become its effective subobject, with A as the effective class, so that the instance may be treated as an instance of A. This change from one effective subobject to another is known as casting. The actual class of an instance never changes.

In answering just these three questions, each of which deals with static properties of the hierarchy, we are able to strip away an enormous amount of complication, including access control, method values, instance-variable values, and the physical layout of instances. We consider it essential, however, that the features that genuinely complicate the inheritance model—multiple inheritance, subobject integrity, shared and non-shared

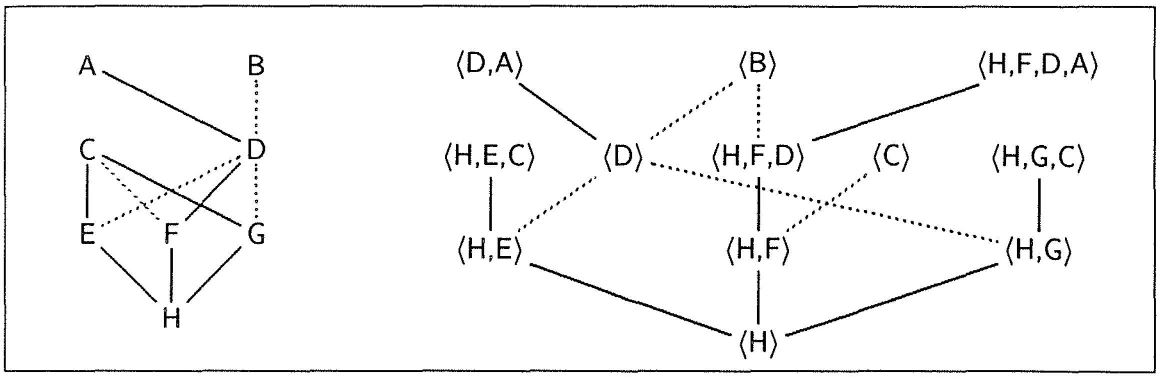


Figure 1: A class-name DAG (left) and its derived subobject poset (right). The dotted and solid lines denote the shared and proprietary inheritance relations. The class at the bottom of an arc inherits from that at the top.

classes, virtual and non-virtual methods, and ambiguity analysis—are retained. Despite our formal simplifications, and partly thanks to them, we have found this model to be an invaluable aid in our design and implementation of a mostly-static multiple inheritance object system with first-class classes[16].

As Snyder suggests, modeling these features entails a certain amount of complexity. It must be remembered that the complexity is not artificially introduced by the formalism, but that the formalism is only as complicated as necessary to model its complex subject. Moreover, we do not present a critique of the inheritance model; rather, the aim is to find a formalism for the existing model as we understand it.

The rest of this paper proceeds as follows: Section 2 lays the conceptual groundwork as it answers Question 1. Questions 2 and 3 are answered in Section 3. Section 4 gives a detailed account of the relation of this model to C++, and Section 5 demonstrates the use of this formalism as a basis for modifications to the inheritance model. Sections 6 and 7 summarize our contribution in relation to previous work. Finally, the appendix proves a key lemma.

2 The Set of Subobjects

Our formalism is specified partly in terms of posets (partially ordered sets). Certain conventions are used: a poset may be described as $\langle A; R\rangle$ for a set A and an order R, where R is reflexive, antisymmetric and transitive; unless otherwise specified, a subset of a poset is a poset with the same order restricted to the elements of the subset. See, for example, Davey and Priestly[11].

Throughout our discussion, we shall make reference to Figure 1, which shows both a class hierarchy (left) and the derived poset of subobjects (right). In each case $x < y$ implies that x appears lower than y, and that x inherits from (is derived from) y. This class-name graph is artificially complex in order that the reader may consider the effects of unusual inheritance relations and attempt to derive the subobject poset from our specification.

Finally, we use $f[y/x]$ to indicate the functional extension of f such that x maps to y.

2.1 Subobject Intuitions

Let us first informally approach the answer to Question 1. We can think of each class as having a unique name that denotes the class itself. Each class definition includes the specification of a set of *shared* base classes and a set of *proprietary* base classes. For example, in Figure 1, F has one proprietary base class D and one shared base class C.

For any class C we may readily construct a DAG whose arcs are a subset of the disjoint union of the shared and proprietary inheritance relations over the class hierarchy; we call this the *class-name graph* for the class C. Note that the shared or proprietary attribute is associated with the arcs, not the nodes. Thus, C is both shared (with respect to F) and proprietary (with respect to E and G.)

Unfortunately, the class-name graph is not an especially useful way of understanding the *results* of the different kinds of inheritance. Specifically, the class-name graph is not suitable for answering our three questions about subobjects. For this purpose we derive the more useful *subobject poset.*

Sakkinen, who also recognizes the need to distinguish between the class-name graph and his DAG equivalent of the subobject poset, informally describes the subobject graph [28](p.80): "The correspondence between *paths* in the two graphs is one-to-one. However, a class in the inheritance graph may correspond to more than one node in the subobject graph, depending on the sharabilities." One of our goals is to model the effects of such sharing.

To develop intuitions about the set of subobjects, consider the following fixed-point algorithm for determining the set of subobjects for an instance of H in Figure 1. Let $\mathcal{C}$ be the set of class names in Figure 1.

1. Initialize the roots set to be $\{\mathsf{H}\}$; initialize the derived set to be $\{\}$.
2. For every element of the roots set, encode every *proprietary* path that reaches any other class in $\mathcal{C}$. Paths are encoded as tuples of class names, such as $\langle \mathsf{H}, \mathsf{F}, \mathsf{D}, \mathsf{A} \rangle$, where H is the root class-name and A is the *reached* class-name. Add each such tuple to the derived set.
3. Whenever a root class-name or a reached class-name has a *shared* arc to another class name, that other class name is added to the root set.
4. Repeat 2 & 3 until both the root set and the derived set reach a fixed point. The final set of subobject labels is given by the union of the derived set with the set of singleton-tuples of the elements of the root set.

In our example, H reaches C through two distinct proprietary paths: $\langle \mathsf{H}, \mathsf{E}, \mathsf{C} \rangle$ and $\langle \mathsf{H}, \mathsf{G}, \mathsf{C} \rangle$. H also reaches F, which has a shared arc to C. This gives us $\langle \mathsf{C} \rangle$, another subobject corresponding to C. These are encoded with different labels to represent the fact that instances of H will have three subobjects corresponding to C.

Thus we identify each subobject with the portion of its path that uniquely specifies its derivation. To model sharing, subobjects are distinguished only by the subpath of exclusively proprietary arcs from the reached class down to either the instantiated class or the first shared arc, as demonstrated above. It is this subpath that constitutes a *subobject label.*

Finally, note that subobject labels are derived with respect to the instantiation of a particular class. Again referring to Figure 1, if the root set were initialized to $\{\mathsf{E}\}$ instead of $\{\mathsf{H}\}$, the set of subobject labels would be $\{\langle \mathsf{E} \rangle, \langle \mathsf{E}, \mathsf{C} \rangle, \langle \mathsf{D} \rangle, \langle \mathsf{D}, \mathsf{A} \rangle, \langle \mathsf{B} \rangle\}$.

2.2 Subobject Formalism

The following formalism is specified in terms of two essential constructs: the *class context* and the *subobject.* Both of these are defined over a domain of class names $\mathscr{C}$ and a domain of member names $\mathscr{M}$.

Definition 2.1 (class context) *A class context γ is a 4-tuple $\langle \mathcal{C}, \nu, \prec_s, \prec_p \rangle$, such that*

$$\mathcal{C} \subseteq \mathscr{C}$$
$$\nu \in \mathcal{C} \to 2^{\mathscr{M}}$$
$$\prec_s, \prec_p \subseteq \mathcal{C} \times \mathcal{C}$$

where the reflexive and transitive closure of the union of $\prec_s$ and $\prec_p$ is antisymmetric.

For each class context γ, we define $<_s = (\prec_s)^+$, $\leq_s = (\prec_s)^*$, $<_p = (\prec_p)^+$, and $\leq_p = (\prec_p)^*$. Similarly, $\prec_{sp} = (\prec_s \cup \prec_p)$, $<_{sp} = (\prec_{sp})^+$, and $\leq_{sp} = (\prec_{sp})^*$.

As a notational convention, $\mathcal{C}$, ν, $\prec_s$, $\prec_p$ (and the aforementioned transitive closures) will refer to the corresponding components of γ, when γ is clear from context.

A class context γ comprises the set of class names $\mathcal{C}$, the function from class names to their member names ν, the set of shared arcs $\prec_s$, and the set of proprietary arcs $\prec_p$. The antisymmetry of $\leq_{sp}$ corresponds to the requirement that class-name graphs be acyclic. As a result, $\langle \mathcal{C}; \leq_{sp} \rangle$ is a poset.

Definition 2.2 (subobject) *A subobject σ is a triple $\langle \gamma, C, \langle X, Y_1, \ldots, Y_n \rangle \rangle$ where γ is a class context, $n \geq 0$, and*

(1) $C, X, Y_1, \ldots, Y_n \in \mathcal{C}$

(2) $X \prec_p Y_1, \prec_p \cdots \prec_p Y_n$

(3) $(C = X) \vee \exists (Z \in \mathcal{C})[C \leq_{sp} Z <_s X]$

Condition 1 simply ensures that the class name C and the subobject label $\langle X, Y_1, \ldots, Y_n \rangle$ consist of names from the class-name graph. Condition 2 uses $\prec_p$ to ensure that each class in the subpath is named in the label; for example, $\langle \mathsf{H}, \mathsf{D} \rangle$ is *not* a subobject label in Figure 1 because it omits the intermediate F. The final condition specifies that if X is not C, it must be at the top of a shared arc whose bottom is an ancestor of C.

A subobject σ, then, consists of a class context γ, an *actual class name* C, and an effective subobject label of the form κZ, where κ is a possibly empty sequence of class names. We call Z the *effective class name* of the subobject, which may also be referred to by $\mathrm{eff}(\sigma)$.

Now we may answer Question 1. We use Σ to refer to the set of all subobjects over $\mathscr{C}$ and $\mathscr{M}$. Then $\Sigma[\gamma]$ refers to that subset of Σ restricted to the subobjects with γ as the first component, and $\Sigma[\gamma, C]$ to that subset of $\Sigma[\gamma]$ in which C is the second component. Thus, $\Sigma[\gamma, C]$ is the set of subobjects for C in γ. This replaces the intuitive fixed-point algorithm from Section 2.1. Whereas the fixed-point algorithm determined the correct subobject labels, here we determine the correct subobjects.

We call subobjects of the form $\langle \gamma, C, \langle C \rangle \rangle$ *primary*; all others are called *dependent*. The primary subobject corresponds to an uncast instance of C, while dependent subobjects correspond to cast instances.

Definition 2.3 (Obj) *Define* Obj *to be the subobject*

$$\mathrm{Obj} \underset{\mathrm{def}}{=} \langle \langle \{\mathsf{Root}\}, \emptyset, \emptyset, \emptyset \rangle, \mathsf{Root}, \langle \mathsf{Root} \rangle \rangle$$

Obj is minimal in the following sense: a subobject must have at least one class in its subobject label—we choose Root. This class and the actual class must be defined in the enclosed γ, and may be the same class—as in our case. Thus, the only restriction on γ is that Root be a member of $\mathcal{C}$—our γ contains no other information.

We now consider an operation for introducing class definitions into existing hierarchies. As is the case for most of the operations in this formalism, this operation is a function from subobjects to subobjects. The *inherit* operation introduces a fresh class into a class context and returns its primary subobject.

Definition 2.4 (inherit) *Let $C \in \mathscr{C}$ and $\mathcal{N} \subseteq \mathscr{M}$, and let $\mathcal{S}, \mathcal{P} \subseteq \mathscr{C}$ be disjoint. Define* $\mathrm{inherit}(C, \mathcal{N}, \mathcal{S}, \mathcal{P})$ *to be the function φ such that, for any subobject $\sigma \in \Sigma[\gamma]$ with $C \notin \mathcal{C}$,*

$$\varphi(\sigma) \underset{\mathrm{def}}{=} \langle \gamma', C, \langle C \rangle \rangle$$

where γ' is the class context

$$\langle \mathcal{C} \cup C, \prec_s \cup (C \times \mathcal{S}), \prec_p \cup (C \times \mathcal{P}), \nu[\mathcal{N}/C] \rangle$$

As long as antisymmetry holds, γ' is clearly a class context. Antisymmetry is ensured because γ' only introduces arcs from C to elements of $\mathcal{C}$, and $C \notin \mathcal{C}$. As a result, $\langle \gamma', C, \langle C \rangle \rangle$ is clearly a subobject in $\Sigma[\gamma', C]$.

The function φ extends σ's class context to include the new class name C with its associated member names $\mathcal{N}$, shared bases $\mathcal{S}$, and proprietary bases $\mathcal{P}$. The result is the primary subobject of C. Note that φ is a partial function from subobjects to subobjects; it is undefined when C is already a class name in σ's class context.

3 Name Resolution

The answers to Questions 2 and 3 involve the member names associated with various classes. We first show how the subobject poset for a given class is derived from the class-name graph. We then show how to determine the *family* of subobjects associated with a given member name. If the family is empty, the reference is invalid, and no subobject is selected. If the family has a greatest lower bound, that subobject is selected. Otherwise, the reference is ambiguous, and no subobject is selected.

3.1 Intuitions for Subobject Selection

For illustration, let us add members to some of the classes in Figure 1. In fact, let us define the class hierarchy from Figure 1 using *inherit*.

$$\begin{aligned}\varphi_{\mathsf{H}} = {} & \mathrm{inherit}(\mathsf{H}, \emptyset, \emptyset, \{\mathsf{E}, \mathsf{F}, \mathsf{G}\}) \\ & \circ\, \mathrm{inherit}(\mathsf{G}, \{\mathsf{q}\}, \{\mathsf{D}\}, \{\mathsf{C}\}) \\ & \circ\, \mathrm{inherit}(\mathsf{F}, \{\mathsf{b}\}, \{\mathsf{C}\}, \{\mathsf{D}\}) \\ & \circ\, \mathrm{inherit}(\mathsf{E}, \{\mathsf{p}\}, \{\mathsf{D}\}, \{\mathsf{C}\}) \\ & \circ\, \mathrm{inherit}(\mathsf{D}, \emptyset, \{\mathsf{B}\}, \{\mathsf{A}\}) \\ & \circ\, \mathrm{inherit}(\mathsf{C}, \{\mathsf{b}\}, \emptyset, \emptyset) \\ & \circ\, \mathrm{inherit}(\mathsf{B}, \{\mathsf{p}\}, \emptyset, \emptyset) \\ & \circ\, \mathrm{inherit}(\mathsf{A}, \emptyset, \emptyset, \emptyset)\end{aligned}$$

where $\circ$ indicates composition: $(f \circ g)(x) = f(g(x))$.

In effect, the application $\varphi_{\mathsf{H}}(\mathrm{Obj})$ is analogous to instantiating H in the specified hierarchy. If b is an instance variable, which subobject of an H instance provides the value when b is referenced? If p is a method name, which subobject of an H instance provides the value when p is called?

Somewhat surprisingly, the formalism does not directly discriminate between instance variables and methods. The name resolution scheme for instance variables and non-virtual methods is identical. We call this *static* resolution, since it is determined relative to the effective class of σ, which is a static property of the instance when static types are available, as in C++. Name resolution for virtual methods depends on the actual class of the subobject—a dynamic property—so we call this *dynamic* resolution. Despite the dynamic nature of virtual methods, implementations are able to use static analysis to eliminate run-time searches; we detail such a strategy in Section 4.

By grouping all member names into a single set, we eliminate any distinction based on the name itself. Rather, we provide two operations for referencing members: *dyn* for dynamic (virtual) method references, and *stat* for the other (static) references. In both kinds of references, the *dominance* rule[15] is used to help disambiguate common inheritance situations that arise with shared base classes. Intuitively, when a shared base defines a member that is later redefined along one path, but not along others, the derived class may unambiguously reference the member as if only the modified path existed.

3.2 Formal Subobject Selection

We begin by defining an order relation for the subobjects of any given class. This is used to derive the subobject poset from the class-name graph.

Definition 3.1 ($\leq_{\mathrm{so}}$) *Let $\sigma, \sigma' \in \Sigma[\gamma, C]$ such that $\sigma = \langle \gamma, C, \kappa \rangle$ and $\sigma' = \langle \gamma, C, X\kappa' \rangle$. Then $\sigma \prec_{so} \sigma'$ iff either $X\kappa' = \kappa Z$ for some $Z \in C$ or $\mathrm{eff}(\sigma) \prec_s X$. Define $\leq_{\mathrm{so}} = (\prec_{so})^*$.*

The first disjunct admits the case where the two subobjects correspond to classes joined by a proprietary arc, such as $\langle \mathsf{H} \rangle$ and $\langle \mathsf{H}, \mathsf{F} \rangle$. The second disjunct admits the case where the corresponding classes are joined by a shared arc, such as $\langle \mathsf{H}, \mathsf{E} \rangle$ and $\langle \mathsf{D} \rangle$.

Lemma 3.2 *$\langle \Sigma[\gamma, C]; \leq_{\mathrm{so}} \rangle$ is a poset, called the subobject poset of C in γ.*

The proof is outlined in an appendix.

3.2.1 Families

Let us now return to our example, φ_{H}. Given the subobject $\varphi_{\mathsf{H}}(\mathrm{Obj})$, which subobject is selected by p? As we alluded previously, this depends in part on whether the reference to p is static or dynamic, but we shall see that this distinction disappears in the case of a primary subobject, such as $\varphi_{\mathsf{H}}(\mathrm{Obj})$. Subobject selection is based on the different paths leading from $\varphi_{\mathsf{H}}(\mathrm{Obj})$ to a subobject whose effective class contains a definition of p. From the definition of φ_{H}, we see that B and E provide definitions of p to H. Each of these classes yields one subobject in H: $\langle \mathsf{B} \rangle$ and $\langle \mathsf{H}, \mathsf{E} \rangle$ respectively; thus we have only these two subobjects to choose from. Of these, $\langle \mathsf{H}, \mathsf{E} \rangle$ is selected because it is the greatest lower bound of the family $\{\langle \mathsf{B} \rangle, \langle \mathsf{H}, \mathsf{E} \rangle\}$. Intuitively, this is because there exists a path through $\langle \mathsf{H}, \mathsf{E} \rangle$ leading to $\langle \mathsf{B} \rangle$.

Definition 3.3 (fam) *Let $a \in \mathscr{M}$.*

$$\mathrm{fam}(\gamma, C, a) \underset{\mathrm{def}}{=} \{\sigma \in \Sigma[\gamma, C] \mid a \in \nu(\mathrm{eff}(\sigma))\}$$

3.2.2 Dynamic References

To achieve the dynamic behavior of virtual methods, we apply the *fam* operation to the actual class of the input subobject. If the resulting subobjects may be ordered by $\leq_{\mathrm{so}}$ to yield a least element, that subobject is unambiguously selected.

Definition 3.4 (dyn) *Let $a \in \mathscr{M}$. Then $\mathrm{dyn}(a)$ is the partial function ϕ from subobjects to subobjects such that, for any subobject $\sigma \in \Sigma[\gamma, C]$,*

$$\phi(\sigma) \underset{\mathrm{def}}{=} \sigma'$$

where $\sigma' = \mathrm{glb}(\mathrm{fam}(\gamma, C, a))$ whenever the greatest lower bound exists.

Thus, for example, if $\langle \gamma, \mathsf{H}, \langle \mathsf{H} \rangle \rangle = \varphi_{\mathsf{H}}(\mathrm{Obj})$ then $\mathrm{dyn}(\mathsf{f})(\langle \gamma, \mathsf{H}, \langle \mathsf{D}, \mathsf{A} \rangle \rangle)$ is still $\langle \mathsf{H}, \mathsf{E} \rangle$. Since the only difference between the primary and dependent subobjects is the effective subobject label, which plays no part in the resolution of dynamic references, every dependent subobject will give the same result as the primary subobject.

3.2.3 Static References

Whereas dynamic references are resolved with respect to the actual class, static references are resolved with respect to the effective class. This is partly a matter of what information is provided to *fam*, and partly a matter of what is done with the result. It is straightforward, using *fam*, to determine a family of subobjects with respect to the effective class, but the resulting set will be a subset of the subobject poset of the effective class, whereas we ultimately require a subobject from the subobject poset of the actual class.

Consider the case of resolving b with respect to H's $\langle \mathsf{H}, \mathsf{E} \rangle$ subobject. The effective class would be E, so *fam* would yield $\{\langle \mathsf{C} \rangle\}$, E's $\langle \mathsf{C} \rangle$ subobject. The problem is that this is isomorphic to H's $\langle \mathsf{H}, \mathsf{E}, \mathsf{C} \rangle$ subobject, not its $\langle \mathsf{C} \rangle$ subobject.

Thus we must translate the selected subobject of the effective class into the corresponding subobject of the actual class. This is certainly possible, since the poset of the effective class is clearly isomorphic to a subset of the poset of the actual class by virtue of subobject integrity.

Definition 3.5 (trans) *Let $\sigma, \sigma' \in \Sigma[\gamma]$ such that $\sigma = \langle \gamma, C, \kappa \rangle$ and $\sigma' = \langle \gamma, \mathrm{eff}(\sigma), X\kappa' \rangle$.*

$$\mathrm{trans}(\sigma, \sigma') \underset{\text{def}}{=} \langle \gamma, C, \kappa''\kappa' \rangle, \textit{ where}$$

$$\kappa'' = \begin{cases} \kappa & \textit{if } \mathrm{eff}(\sigma) = X, \\ X & \textit{otherwise.} \end{cases}$$

Lemma 3.6 *The result of the* trans *operation is a subobject.*

This is easily shown by recognizing that the two cases for κ'' correspond to conditions 2 and 3 in Definition 2.2, respectively.

Definition 3.7 (stat) *Let $a \in \mathscr{M}$. Then* $\mathrm{stat}(a)$ *is the partial function ψ from subobjects to subobjects such that, for any $\sigma = \Sigma[\gamma]$,*

$$\psi(\sigma) \underset{\text{def}}{=} \mathrm{trans}(\sigma, \sigma')$$

where $\sigma' = \mathrm{glb}(\mathrm{fam}(\gamma, \mathrm{eff}(\sigma), a))$ whenever the greatest lower bound exists.

Thus, for example, $(\mathrm{stat}(\mathsf{b}) \circ \mathrm{dyn}(\mathsf{p}) \circ \mathrm{dyn}(\mathsf{q}) \circ \varphi_{\mathsf{H}})(\mathrm{Obj})$ can be determined to select H's $\langle \mathsf{H}, \mathsf{E}, \mathsf{C} \rangle$ subobject. The virtual method calls to q and then to p lead to the selection of the $\langle \mathsf{H}, \mathsf{E} \rangle$ subobject. From there, the reference to b yields the $\langle \mathsf{H}, \mathsf{E}, \mathsf{C} \rangle$ subobject as discussed earlier. Note, however, that $(\mathrm{stat}(\mathsf{b}) \circ \varphi_{\mathsf{H}})(\mathrm{Obj})$ is ambiguous, since the minima of $\{\langle \mathsf{H}, \mathsf{E}, \mathsf{C} \rangle, \langle \mathsf{H}, \mathsf{G}, \mathsf{C} \rangle, \langle \mathsf{C} \rangle, \langle \mathsf{H}, \mathsf{F} \rangle\}$—$\langle \mathsf{H}, \mathsf{E}, \mathsf{C} \rangle$, $\langle \mathsf{H}, \mathsf{F} \rangle$, and $\langle \mathsf{H}, \mathsf{G}, \mathsf{C} \rangle$—are incomparable.

4 Relation to C++

Since C++ is far and away the most widely-known exemplar of this style of object-oriented programming language, we consider how our model relates directly to C++. Due to the informal specification of C++, we attempt no formal proof of this relationship. As Perlis notes, "One can't proceed from the informal to the formal by formal means."[24].

A great deal of the complication in a C++ compiler concerns the management of virtual methods and virtual classes. Part of this relates to the determination of the layout of an instance of a given class; although we do not address the physical layout of an instance, we do provide the necessary set of subobjects for the instance. The greater complication is subobject selection as it relates to the resolution of method calls and instance-variable references; our formalism provides a useful model for resolving these questions.

Of particular interest is the case of pointers or references to objects, which are subject to *subsumption*, by which a variable declared as a pointer(reference) to instances of one class may serve as a pointer(reference) to instances of any class for which it is an ancestor. Method inheritance is supported in this way by forcing the hidden method-parameter `this` to be such a pointer.

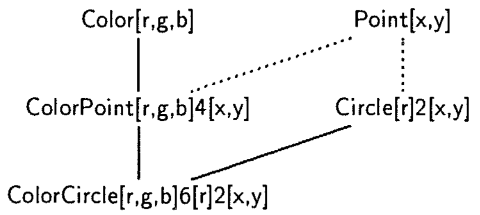
Compiled code must not make broad assumptions about the layout of an instance when that instance is subject to subsumption. This becomes especially apparent in the case of multiple inheritance with virtual classes and virtual methods, where only a limited set of references and method calls can be made without the aid of some indirection, either through a *virtual method table* (a *vtbl*) or a similar construct for virtual classes.

We now consider in some detail the role of subobject selection in the compilation of method calls and instance-variable references. In each case, the instance that is used for the reference can be thought of as a subobject in our model. The effective class of this subobject is known at the call site (or reference site) at compile time, but not the actual class (due to subsumption.) For simplicity, we assume the instance is reached by a pointer variable, `i`, although a reference variable would work similarly. In the following, let σ be the subobject denoted by `i`, and let C be the effective class of σ. Also for simplicity, we assume instances are contiguous sequences of subobjects.

4.1 C++ Static References

References to instance variables `i->`a can be compiled to references into the given instance at a particular offset from either the start of the current subobject or the start of a shared base subobject. As specified in the *stat* operation, the effective class C of σ is used as the basis for a search that yields σ', a subobject of C with an unambiguous definition of a. Unless σ' is a shared subobject of C, the value of a may be found at a fixed offset from the starting address of σ. The compiled code accesses the dynamic instance at this fixed offset.

If σ' is a shared subobject of C, the offset to the member is not fixed, since multiple inheritance cannot preserve the ordering of shared subobjects. Rather, the reference must be indirect through a pointer or offset *found* at a fixed offset. For example:



This is a class-name graph deriving a ColorCircle class. Each class is annotated with a depiction of its instances, with the square brackets marking the subobjects. Notice that, in order to support subsumption, none of the classes that inherit from the shared base class can assume the location of the corresponding subobject. Rather, they assume the location of an offset to the start of the subobject.

Thus, a site that references the x field of a ColorPoint is compiled to find an offset at index 3 of the current subobject. That offset locates the address of the selected subobject, where the value of x is found at index 0. This same compiled code still works for a ColorCircle, even though the Point subobject is at a different relative address. Similarly, a Circle instance must store the offset at index 1. Thus a ColorCircle that has been cast to its Circle subobject also has an offset at index 1. In this case, the offset is calculated from the beginning of the Circle subobject of the ColorCircle instance.

Method calls `i->`a`(`$x_1, \ldots, x_m$`)` are quite similar when the method is not virtual. The subobject is determined exactly the same way. The starting address of σ' relative to the starting address of σ can be calculated and used to cast σ as it is passed to the function. The address of the method function can be obtained statically as well: we presuppose the existence of a total function *methfun*:$(\mathscr{C} \times \mathscr{M}) \rightarrow$ *methfuns*. The method address is obtained by methfun(eff(σ'), a).

In each case, the physical layout of each instance ensures that the subobject address calculated for the effective class yields the correct subobject for any derived-class instance that is cast to the effective class. In the case of shared subobjects, this means arranging for the run-time indirection.

4.2 C++ Dynamic References

Dynamic references correspond to virtual methods in C++. Unlike a static method call, a virtual method call cannot be fully statically resolved. Due to subsumption, resolution of the call is based on the actual class of the instance rather than the effective class, but only the effective class is known at compile time.

Resolution of the call `i->`a`(`$x_1, \ldots, x_m$`)` amounts to finding both the method function and the casting address with respect to a particular run-time value of `i`. This is accomplished *without* a run-time search by the use of indirection through a *virtual method table* (a *vtbl*) in the instance, which is shared by all instances of the same class. Subsumption is managed by arranging the vtbl of a derived class to have the same shape as that of the base class; that is, the dynamic method information for a method a is found at the same address in both vtbls.

Because of multiple inheritance, the same instance may be cast and used in contexts that expect a differently-shaped vtbl. For example, given a class C derived multiply from A and B, if A's vtbl is ordered $[f, g, h]$ and B's is ordered $[g, f]$, how can C's vtbl conform to both? The solution is to associate a different vtbl with each subobject, corresponding in shape to the ordering of the effective class of the subobject. In our example, C would have a $[f, g, h]$ vtbl for its $\langle C, A\rangle$ subobject and a $[g, f]$ vtbl for its $\langle C, B\rangle$ subobject. For the primary subobject, an arbitrary ordering may be used since no conformance is required. Typically, however, the $\langle C\rangle$ and $\langle C, A\rangle$ vtbls would be shared as part of an overall subobject compression scheme, with C's new methods appended to the end. Semantically, this is just a special case of an arbitrary ordering.

Using this technology, we must answer two questions regarding virtual methods: how is the call site compiled, and how are the vtbls arranged? For simplicity, assume all $a \in \mathscr{M}$ are virtual method names, and each redefinition of a method a is fully congruent (identical in its formal parameter types and return type) with the version(s) it supersedes. This ensures that every method definition is either the original definition of a virtual method or a safe overriding definition.

We associate an ordering of methods with each class, corresponding to the shape of the vtbl of its primary subobject. First we identify the set of method names associated with the class.

Definition 4.1 (reach) *Given a class context γ with $C \in \mathcal{C}$,*

$$\text{reach}(\gamma, C) \underset{\text{def}}{=} \bigcup_{D \in \{\widehat{C} \mid C \leq_{sp} \widehat{C}\}} \nu(D)$$

Then we define the (arbitrary) ordering of these names for the primary subobject.

Definition 4.2 (pri) *Let $R = \text{reach}(\gamma, C)$ and let n be its cardinality. Then*

$$\text{pri}(\gamma, C) \underset{\text{def}}{=} \langle a_1, \ldots, a_n\rangle$$

where $a_1, \ldots, a_n \in R$.

4.2.1 Virtual Method Tables

We next associate a vtbl with each subobject. The function *vtbl* maps subobjects to their associated virtual method tables.

Definition 4.3 (vtbl) *Let $\langle a_1, \ldots, a_n\rangle = \text{pri}(\gamma, \text{eff}(\sigma))$, where $\sigma \in \Sigma[\gamma, C]$. Then for $1 \leq i \leq n$,*

$$\sigma_i = \text{dyn}(\sigma)(a_i)$$
$$\mu_i = \text{methfun}(\text{eff}(\sigma_i), a_i)$$
$$\Delta_i = \&\sigma_i - \&\sigma$$

Then $\text{vtbl}(\sigma) \underset{\text{def}}{=} \langle(\mu_1, \Delta_1), \ldots, (\mu_n, \Delta_n)\rangle$.

Similar to the C++ usage, we use & to denote the address of a subobject. Note that the use of *dyn* guarantees the actual class of σ is used to find the casting information, while the use of *eff*(σ) ensures that the ordering of the vtbl conforms to the effective class. As a result, the same method name maps to the same virtual-method information in every vtbl of every subobject of the same class. The different method tables of the same instance differ only in the subset of the methods that they define and the order in which they are represented. Note, also, that ambiguities are detected at vtbl-creation time, not at run time.

4.2.2 Virtual Method Call Sites

At the call site, `i->`$a(x_1, \ldots, x_m)$, the vtbl ordering is known to be $\text{pri}(\gamma, C) = \langle a_1, \ldots, a_n\rangle$, for some γ. If there exists a $k \leq n$ such that $a = a_k$, the method is reachable and the static vtbl-index is k. Otherwise the call is statically determined to be invalid.

A valid call-site is compiled so that, at run time, a (μ_k, Δ_k) pair is extracted from the vtbl of the dynamic instance. A new instance-pointer is cast from `i` using Δ_k. Then the method function μ_k is invoked using this pointer and the $x_1, \ldots, x_m$ arguments.

4.3 Dominance

In the simplest view of ambiguity, a member is ambiguous whenever two distinct subobjects exists that both define the member such that a path exists to one subobject that does not pass through the other. The dominance rule modifies this view, providing a useful disambiguation for a common name-conflict in multiple-inheritance systems.

When a shared base class results in a subobject that is reached by more than one path, it is possible that one or more of these paths may contain overriding definitions of a member of the shared subobject. According to the dominance rule, a member is not ambiguous simply because an overridden definition is also accessible along a path that does not contain an overriding definition.

Since we organize subobjects into a poset rather than a DAG, we do not have a formal notion of a path. In fact, certain path information is lost in the transformation of a DAG to a poset. Even worse, it almost appears to be exactly the kind of information that is required to model dominance. Consider the example:

$$\begin{aligned} \varphi = {} & \mathrm{inherit}(\mathsf{C}, \{\mathsf{A},\mathsf{B}\}, \emptyset, \emptyset) \\ & \circ\, \mathrm{inherit}(\mathsf{B}, \emptyset, \{\mathsf{x}\}, \{\mathsf{A}\}) \\ & \circ\, \mathrm{inherit}(\mathsf{A}, \emptyset, \{\mathsf{x}\}, \emptyset) \end{aligned}$$

Here, the subobject graph would be isomorphic to the class-name graph, preserving the fact that C reaches A by two paths—directly, and through B. The subobject poset, however, records only the fact that $\langle \mathsf{C} \rangle \leq_{\mathsf{so}} \langle \mathsf{C}, \mathsf{B} \rangle \leq_{\mathsf{so}} \langle \mathsf{A} \rangle$. The information that C reaches A directly has been lost.

Rather than causing a problem, however, this loss of information fortunately corresponds to the dominance rule. It is exactly this missing path that would have caused an ambiguity in accessing x, and that the dominance rule requires us to ignore. Since this is just a special case of our requirement that there be a least subobject, there is no need to add an explicit dominance rule to our model. Rather than being a separate case, dominance is just the natural behavior of name resolution.

5 Modifications

An important implication of any abstraction is that it provides a coherent basis for the exploration of related systems. This is true of our formalism as well. As an extension, for example, we might consider a naive privatization mechanism such that privatized members are not visible to derived classes. If we change *inherit* to split the ν field into two fields, ν_e (external public names) and ν_i (internal private names), we may redefine variable references by changing the definition of *fam*.

Definition 5.1 (fam (revised)) *Let* $a \in \mathscr{M}$.

$$\mathrm{fam}(\gamma, C, a) \underset{\mathrm{def}}{=} \left\{ \sigma \in \Sigma[\gamma, C] \;\middle|\; \begin{array}{l} a \in \nu_e(\mathrm{eff}(\sigma)) \ \mathit{or} \\ \mathrm{eff}(\sigma) = C \ \&\ a \in \nu_i(C) \end{array} \right\}$$

This redefinition applies protection attributes as a visibility mask, unlike C++ but as suggested by Sakkinen[28]. As an example, consider a modified φ_{H} in which all the existing members were made public except the b member of C. Then b would no longer be ambiguous in H since the only reachable definition would be in $\langle \mathsf{H}, \mathsf{F} \rangle$.

6 Related Work

The problem of combining multiple hierarchies has been dealt with in many ways. Snyder[32] divides these into linear and graph-oriented approaches. Linear approaches, such as [1, 23, 12, 13, 14], do not model subobject integrity. Graph-oriented approaches, all of which are capable of supporting subobject integrity in some way, include extended Smalltalk[2], Trellis/Owl[29], CommonObjects[31], ROME[7], and the Krogdahl/Stroustrup model discussed here.

Some formal models of graph-oriented multiple inheritance systems[20, 36] have been based on class-name posets rather than class-name graphs. As discussed earlier, the transition from graph to poset can lead to information loss. At the class-name level, this loss of path information leads to the collapsing of some distinct subobjects, and is insufficient to model the kind of inheritance we have formalized.

Cardelli's model of multiple inheritance[3, 5, 4] is closely tied to a record representation of objects, and does not attempt to maintain subobject integrity; similarly with Compagnoni and Pierce[25, 8]. In effect, the same may be said for Eiffel[18, 22]: distinct storage is maintained only if the programmer renames all members to protect against conflicts. The only difference is that Eiffel forces the renaming, while a record semantics tends to allow collapsing.

Snyder's model of the C++ object model[33] offers a more comprehensive modeling of the C++ object system in general, but in doing so it explicitly ignores a number of complications that we treat here, especially the problem of repeated inheritance. Seligman[30] presents another formal semantics of C++, but makes no attempt to deal with multiple inheritance. Wallace's semantics of C++[37] includes multiple inheritance, but with virtually no concern for compile-time issues such as subobjects, subobject selection, and ambiguity analysis.

7 Conclusion

Our new formal model provides an implementation-independent means of understanding the complex interaction of features in a particular variety of multiple inheritance. Natural-language descriptions, buried among the other details of a specific language, lack the rigor and accessibility of a formal specification. For example, an implementation can never be provably correct with respect to an informally specified semantics. A model such as ours provides a more precise reference for both implementors and language theorists.

Using our formalism, sample hierarchies may be constructed and formally analyzed for certain properties, such as the effects of arbitrary sequences of method invocations and the reachability or ambiguity of specific instance variables or methods. This, in turn, opens the door to automated static analysis tools. Moreover our model forms the basis for an extended formal study of related inheritance systems so that modifications may be evaluated in the absence of implementations.

8 Acknowledgment

The ideas that led to this paper were greatly shaped by our conversations with Michael Ashley, Shinnder Lee, Anurag Mendhekar, and Jonathan Sobel, all of whose continuing input has been invaluable. Thanks, also, to Markku Sakkinen for his thought-provoking suggestions for improvements of two drafts. We also thank Mitch Wand for his insightful critique. Finally, we thank the anonymous OOPSLA reviewers, whose helpful comments had a great influence on the final form of this paper.

References

[1] BOBROW, D. G., KAHN, K., KICZALES, G., MASINTER, L., STEFIK, M., AND ZDYBEL, F. CommonLoops: Merging Lisp and object-oriented programming. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices* (1986), pp. 17–29.

[2] BORNING, A., AND INGALLS, D. Multiple inheritance in Smalltalk-80. In *Proceedings AAAI '82* (1982), pp. 234–237.

[3] CARDELLI, L. A semantics for multiple inheritance. In *Semantics of Data Types*, G. Kahn, D. B. MacQueen, and G. Plotkin, Eds., LNCS 173. Springer-Verlag, New York, 1984, pp. 51–67.

[4] CARDELLI, L. A semantics of multiple inheritance. *Information and Computation 76* (1988), 138–164. Special issue devoted to *Symp. on Semantics of Data Types*, Sophia-Antipolis (France), 1984.

[5] CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction, and polymorphism. *Computing Surveys 17*, 4 (1985), 471–522.

[6] CARGILL, T. A. Controversy: The case against multiple inheritance in C++. *Computing Systems 4*, 1 (1991), 69–82.

[7] CARRÉ, B., AND GEIB, J.-M. The point of view notion for multiple inheritance. In *Proceedings OOPSLA-ECOOP '90, ACM SIGPLAN Notices* (1990), pp. 312–321.

[8] COMPAGNONI, A. B., AND PIERCE, B. C. Multiple inheritance via intersection types. Tech. Rep. ECS-LFCS-93-275, University of Edinburgh, 1993. Also Technical Report 93-18, C.S. Department, Catholic University Nijmegen.

[9] COOK, W., AND PALSBERG, J. A denotational semantics of inheritance and its correctness. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices* (1989), pp. 433–443.

[10] COOK, W. R. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989. Technical Report CS-89-33.

[11] DAVEY, B. A., AND PRIESTLY, H. A. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[12] DUCOURNAU, R., AND HABIB, M. On some algorithms for multiple inheritance in object oriented programming. In *Proceedings ECOOP '87* (1987), LNCS 276, Springer-Verlag, pp. 243–252.

[13] DUCOURNAU, R., HABIB, M., HUCHARD, M., AND MUGNIER, M. L. Monotonic conflict resolution mechanisms for multiple inheritance. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices* (1992), pp. 16–24.

[14] DUCOURNAU, R., HABIB, M., HUCHARD, M., AND MUGNIER, M. L. Proposal for a monotonic multiple inheritance linearization. In *Proceedings OOPSLA '94, ACM SIGPLAN Notices* (1994), pp. 164–175.

[15] ELLIS, M. A., AND STROUSTRUP, B. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[16] FRIEDMAN, D. P., HAYNES, C. T., MENDHEKAR, A., AND ROSSIE, JR., J. G. *Scheme++: A Static Object-Oriented Scheme Extension with Multiple Inheritance, v1.0.* Indiana University Dept. of Computer Science, 1995. Work in Progress.

[17] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[18] INTERACTIVE SOFTWARE ENGINEERING, INC. *Eiffel Reference Manual*, August 1989. Technical Report TR-EI-17/RM (version 2.2).

[19] KAMIN, S. Inheritance in SMALLTALK-80: A denotational definition. In *Proceedings POPL '88* (1988), pp. 80–87.

[20] KNUDSEN, J. L. Name collision in multiple classification hierarchies. In *Proceedings ECOOP '88* (1988), LNCS 322, Springer-Verlag, pp. 93–108.

[21] KROGDAHL, S. Multiple inheritance in Simula-like languages. *BIT 25* (1984), 318–326.

[22] MEYER, B. *Eiffel: The Language*. Prentice Hall, 1992.

[23] MOON, D. A. Object-oriented programming with *Flavors*. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices* (1986), pp. 1–8.

[24] PERLIS, A. J. Epigrams on programming. *SIGPLAN Notices 17*, 9 (September 1982), 7–13.

[25] PIERCE, B. C. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, December 1991.

[26] REDDY, U. Objects as closures: Abstract semantics of object-oriented languages. In *Conf. on LISP and Functional Programming* (1988).

[27] SAKKINEN, M. Disciplined inheritance. In *Proceedings ECOOP '89* (1989), The British Computer Society Workshop Series, Cambridge University Press, pp. 39–56.

[28] SAKKINEN, M. A critique of the inheritance principles of C++. *Computing Systems 5*, 1 (1992), 69–110.

[29] SCHAFFERT, C., COOPER, T., BULLIS, B., KILLIAN, M., AND WILPOLT, C. An introduction to Trellis/Owl. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices* (Nov. 1986), pp. 9–16. Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

[30] SELIGMAN, A. FACTS: A formal analysis of C++: Type rules and semantics. B.A. Honors Thesis, Williams College, May 1995.

[31] SNYDER, A. CommonObjects: An overview. *ACM SIGPLAN Notices 21*, 10 (October 1986), 19–28.

[32] SNYDER, A. Inheritance and the development of encapsulated software components. In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Eds. MIT Press, 1987, pp. 165–188.

[33] SNYDER, A. Modeling the C++ object model, an application of an abstract object model. In *Proceedings ECOOP '91* (1991), LNCS 512, Springer-Verlag, pp. 1–20.

[34] STEELE JR., G. L. *Common Lisp: The Language*, 2nd ed. Digital Press, 1990.

[35] STROUSTRUP, B. Multiple inheritance for C++. *Computing Systems 2*, 4 (1989).

[36] TOURETZKY, D. S. *The Mathematics of Inheritance Systems*. Research Notes in Artificial Intelligence. Pitman, 1986.

[37] WALLACE, C. The semantics of the C++ programming language. In *Specification and Validation Methods for Programming Languages*, E. Boerger, Ed. Clarendon Press, Oxford, 1995, pp. 131–163.

Appendix

Lemma 3.2: $\langle \Sigma[\gamma, C]; \leq_{\mathrm{so}} \rangle$ *is a poset, called the subobject poset of* C *in* γ.

$\langle 1 \rangle 1.$ $\leq_{\mathrm{so}}$ is a reflexive, antisymmetric and transitive closure over $\Sigma[\gamma, C]$.

$\langle 2 \rangle 1.$ It suffices to show that $\leq_{\mathrm{so}}$ is antisymmetric, since it is defined by reflexive and transitive closure.

$\langle 2 \rangle 2.$ $\sigma \leq_{\mathrm{so}} \sigma'$ and $\sigma' \leq_{\mathrm{so}} \sigma$ implies $\sigma = \sigma'$.

LET: $\sigma = \langle \gamma, C, X\kappa Z \rangle$ and $\sigma' = \langle \gamma, C, X'\kappa' Z' \rangle$.

ASSUME: $\sigma \leq_{\mathrm{so}} \sigma'$ and $\sigma' \leq_{\mathrm{so}} \sigma$ but $\sigma \neq \sigma'$.

PROVE: False.

$\langle 3 \rangle 1.$ $\leq_{sp}$ is antisymmetric, by Def (2.1).

$\langle 3 \rangle 2.$ $\sigma \leq_{\mathrm{so}} \sigma'$ and $\sigma \neq \sigma'$ implies $Z <_{sp} Z'$.

CASE: $X\kappa$ is a proper prefix of $X'\kappa'$.

$\langle 4 \rangle 1.$ $Z <_p Z'$ by the antisymmetry of $\prec_p$.

CASE: $\exists(Y \in C)[Z \leq_{sp} Y \prec_s X']$

$\langle 4 \rangle 2.$ $Z <_{sp} Z'$, since $X' <_p Z'$.

$\langle 4 \rangle 3.$ Q.E.D.

$\langle 3 \rangle 3.$ $\sigma' \leq_{\mathrm{so}} \sigma$ and $\sigma \neq \sigma'$ implies $Z' <_{sp} Z$. (Proof as above.)

$\langle 3 \rangle 4.$ Contradiction.

$\langle 3 \rangle 5.$ Q.E.D.

$\langle 2 \rangle 3.$ Q.E.D.

$\langle 1 \rangle 2.$ Q.E.D.