Foundations for a Semantics of Subobjects

Extended Abstract

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

{jrossie,dfried}@cs.indiana.edu

Department of Computer Science, Indiana University

215 Lindley Hall, Bloomington, Indiana 47405

May 17, 1995

Abstract

We find that by answering three essential questions about the static properties of inheritance hierarchies, we gain significant insight into the kind of inheritance best exemplified by C++. Multiple inheritance, the dominance rule for disambiguation, and the presence of virtual and non-virtual base classes lead to quite a complicated semantics. By abstracting the notion of a subobject, we have designed an algebra for answering the three questions in a way that manages all of these complications.

Overview

The multiple-inheritance model found in C++[2, 6], which derives largely from Krogdahl's multiple inheritance for Simula[3], can be seen as the result of one fundamental design imperative: that distinct storage should be allocated for each instance variable, regardless of name identity. This design choice, which we refer to as the subobject-integrity imperative, leads to a space of closely-related semantics in which the C++ and Simula models represent two points.

It has been our ongoing project to find a semantic framework for subobject integrity such that the semantics of the C++ model, for example, may be seen as a specialization. The key issue that complicates such a semantics is the ambiguity that can arise in method calls and instance-variable references.

The problem stems from the lack of unique labels. As mentioned earlier, the names of the instance variables may not be unique. The same holds for method names: a method must be associated with the expected instance variables, so name equivalence does not imply method equivalence. Moreover, it is not sufficient to label instance variables and methods with the class name from which they are inherited, since the same class, when reached through different inheritance paths, can result in distinct storage.

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

The solution to the problem of labels is to lift the notion of a subobject out of the implementation and into our semantics. We devise a labeling scheme for subobjects and use these subobject labels as the unique labels for identifying instance variables and methods.

The labeling scheme for subobjects is complicated by the presence of shared inheritance. In C++, virtual classes provide a means of specifying that repeated inheritance from the same base class should not lead to distinct storage in the derived class; that is, it should not lead to separate subobjects. We call this shared inheritance. Without shared inheritance, a subobject may be uniquely specified by an inheritance path of class names; the arcs in the inheritance path are homogeneous. With shared inheritance, however, the arcs are either shared or proprietary, and the labels must only encode those classes in the path that lie above the highest shared arc. (We use the common intuition that a base class is *above* its derived classes.)

Shared classes unfortunately contribute to the complexity of the inheritance model. Cargill[1] makes an ample case for the complexity of the system, as do the various attempts at formalizing the C++ object model, most notably Snyder's[5]. Cargill, arguing against the introduction of Stroustrup's multiple inheritance system into C++, complains that (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. Snyder's model is similar to ours in that it deals with subobjects on an abstract 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."

It is not our purpose to argue for or against the inheritance model, but merely to devise a concise expression of it. 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 the subobject integrity imperative:

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

Question 2 (methods) *For a specific method name and an instance of a given class, which subobject will be cast-to*[1] *as a result of the call? (Or will it be ambiguous?)*

Question 3 (instance variables) *For a specific instance-variable name and an instance of a given class, which subobject will contain the value? (Or will it be ambiguous?)*

[1] Since we have no explicit notion of a type, casting is merely a matter of accessing a particular subobject.

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 protection/privatization, method values, instance-variable values, and even the class/instance distinction. We consider it essential, however, that the features that genuinely complicate the inheritance model—multiple inheritance, shared and non-shared classes, virtual and non-virtual methods, and lack of linearization—are retained. Despite our formal simplifications, and partly thanks to them, we have found this model to be an invaluable aid in our practical understanding of this kind of inheritance[4].

References

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

[2] Ellis, M. A., and Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[3] Krogdahl, S. Multiple inheritance in simula-like languages. *BIT 25* (1984), 318–326.

[4] Rossie, Jr., J. G., and Friedman, D. P. An algebraic semantics of subobjects. In *OOPSLA '95: ACM Conference on Object-Oriented Programming Systems, Languages and Applications (to appear)* (Austin, Texas, October 1995).

[5] Snyder, A. Modeling the C++ object model, an application of an abstract object model. In *Proceedings ECOOP '91: European Conference on Object-Oriented Programming* (Geneva, Switzerland, July 1991), P. America, Ed., LNCS 512, Springer-Verlag, pp. 1–20.

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