Modeling Subobject-Based Inheritance

Jonathan G. Rossie Jr.[1*†] Daniel P. Friedman[1*] Mitchell Wand[2‡]

[1] Department of Computer Science, Indiana University
Bloomington, IN 47405, USA
{jrossie,dfried}@cs.indiana.edu
[2] College of Computer Science, Northeastern University
Boston, MA 02115, USA
wand@ccs.neu.edu

Abstract. A model of subobjects and subobject selection gives us a concise expression of key semantic relationships in a variety of inheritance-based languages. Subobjects and their selection have been difficult to reason about explicitly because they are not explicit in the languages that support them. The goal of this paper is to present a relatively simple calculus to describe subobjects and subobject selection explicitly. Rather than present any deep theorems here, we develop a general calculus that can be used to explore the design of inheritance systems.

1 Introduction

In a system with inheritance, a class **P** (Point) represents a collection of members, the methods and instance variables that are shared by instances of **P**. When a class **CP** (ColorPoint) inherits from **P**, the language may be designed to do one of two things. It may attempt to merge the members of **P** with those of **CP**, usually by collapsing same-named members into single definitions. Alternatively, it may allow the members of **P** to be inherited as an indivisible collection. This collection, when instantiated, is known as a subobject. Each object of class **CP** has a distinct subobject **CP/P**, which we call the **P** subobject of **CP**, as well as a subobject **CP/CP**, which we call the primary subobject of **CP**. Subobjects are meant to support subclass polymorphism: each subobject represents a different view of the object, allowing it to be viewed as an instance of any of its ancestor classes. The notion of subobjects is necessary to model the behavior of a variety of complex multiple-inheritance systems such as in C++.

Since an instance no longer can be seen as a record with member names as field names, member references become more complicated. In this model, it is the subobjects that are seen as records, and an object is just a collection of subobjects. Member references are made by selecting a subobject that defines

*Research partially supported by NSF grant CCR-9302114.

†Current address: Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA

‡Received support from NSF grants CCR-9304144 and CCR-9404646.

the member, and referencing the appropriate field of that subobject. Since the field reference is unremarkable, we focus on the semantics of subobject selection.

In multiple-inheritance systems, it becomes complicated even to determine the set of subobjects for instances of a given class. In some systems, the methods and instance variables of **P** may in fact be *replicated* many times in **CP** depending on the different paths through which **P** may be inherited by **CP**. In other systems, there may be only one subobject for each ancestor, despite multiple paths to the same ancestor. Yet other systems may allow both kinds of inheritance to coexist, as in C++ [10, 26].

Subobject selection is complicated by the multiple views supported by an object. When a **CP** instance is viewed through its **CP/P** subobject, should it still be possible to access the members defined in the **CP/CP** subobject as well? What about the converse? And if both are accessible, which should be preferred if a member is defined in both subobjects? Moreover, how are these decisions affected by the presence of both late-binding (dynamic) and early-binding (static) references? These are the questions our semantics is designed to answer. We develop a core semantics, and extend it to express both single and multiple inheritance. Our goal is to present a general calculus that can be used to explore the design of inheritance systems.

Each of our systems is presented as a language in which an inheritance hierarchy is built and a query is made with respect to the hierarchy. Each query first identifies a primary subobject of some class in the hierarchy, and then asks which subobject of that class is selected when a certain member (or sequence of members) is referenced. An expression in one of these languages is seen as expressing the subobject that is ultimately selected as a result of a sequence of member references. Importantly, we are not concerned with the value of a member reference; we model only static properties of inheritance hierarchies.

For example, suppose we have a new object of class **CP**, and we select (statically) the **f** method, and then from inside that method we select (dynamically) the **g** method. What subobject of **CP** should the **g** method access? This may be expressed as $\mathbf{inh(P;;f,g)\ in\ inh(CP;P;g)\ in\ CP.stat(f).dyn(g)}$, which is a term in our single-inheritance system. Call this term t. The semantics of our system allows us to show that $\vdash t \rhd \mathbf{CP/CP}$, which says that this term selects the primary subobject of **CP**.

Our presentation is expressed in terms of an abstract syntax and an operational semantics for each member of a family of inheritance systems. Since our goal is modeling, we prove no deep theorems about these systems, but we do state some of their rudimentary properties. We begin with a common core in Sect. 2, and extend it to arrive at a single-inheritance system in Sect. 3. Section 4 extends the core in a different direction, arriving at a core system for multiple inheritance. Sections 5 and 6 extend this multiple-inheritance core in two directions, giving us replicating inheritance in one case, and shared inheritance in the other. Finally, Sect. 7 combines the replicating and shared systems into a single system, which closely models the C++ multiple-inheritance system.

2 Class-System Core

We begin with a core system, which we call CSC. Although this is not a powerful system itself, it is the common basis for all the systems that follow. Classes may be defined in CSC, but there is no inheritance, and therefore all subobjects are primary. Thus, subobject selection always yields the primary subobject. Nevertheless, the study of CSC lays important groundwork for the inheritance systems that follow.

2.1 CSC Syntax

A term T makes queries about subobject selection, but may first contribute to the construction of a class context, the inheritance hierarchy in which the queries are resolved. Environments encode the class context, including information about the classes that are defined and the members defined by those classes. The **class** form introduces new classes, but there is no notion of inheritance. The only query in CSC is the trivial a, which simply selects the primary subobject of the named class. Throughout, we let a–d range over class names, while every m is a member name.

CSC Syntax

$T ::=$	CSC term
$\quad \textbf{class}(a; m_1, \ldots, m_n) \textbf{ in } T$	$(n \geq 0)$ class definition
$\quad Q$	query
$Q ::=$	CSC query
$\quad a$	primary subobject of a

For now, a subobject is identified by two classes, although this is extended in the multiple-inheritance systems in later sections. The first class is called the *actual* class of the subobject: In an instance of class a, every subobject's actual class is a. The second class is called the *effective* class of the subobject: Each subobject corresponds to an ancestor of the actual class, possibly the class itself. This ancestor is the effective class of the subobject. Subobjects are annotated as a/b, where a is the actual class and b is the effective class.

Judgments in CSC and the subsequent systems are categorized by the role they play in the proof system. Each of the categories *environments*, *well-formed subobjects*, and *subobject selection* has only one form of judgment. The *access* category comprises all judgments that de-construct environments, while *subobject relations* is a catch-all for the remaining judgment forms, each of which expresses some kind of relationship among subobjects with respect to a given environment.

CSC Judgments

environments	
$\vdash E\ env$	E is a well-formed environment
access	
$E \vdash a\ class$	a is a defined class
$E \vdash a \ni m$	a defines the member m
wf subobjects	
$E \vdash a/b\ wf$	a/b is a well-formed subobject
subobject selection	
$E \vdash T \rhd a/b$	the term T selects a/b

Environments contain information about the class names and member names defined in an inheritance hierarchy.

Environment Syntax

$E ::=$	Environment
$\emptyset$	the empty environment
E, a	definition of class a
$E, a \ni m$	a defines the member m
$E, a \prec b$	a inherits directly from b (SI only)
$E, a \prec_r b$	replicating direct inheritance (RMI, SRMI)
$E, a \prec_s b$	shared direct inheritance (SMI, SRMI)

Sometimes it is useful to view an environment E as a set.

$$\mathrm{dom}(E) = \{a \mid (a \in E) \vee (a \prec b \in E) \vee (a \prec_r b \in E) \vee (a \prec_s b \in E)\}$$
$$\mathrm{codom}(E) = \{b \mid (a \prec b \in E) \vee (a \prec_r b \in E) \vee (a \prec_s b \in E)\}$$

2.2 CSC Rules

The rules for CSC are listed below. This system is intended primarily as a starting point in our derivation of inheritance systems. As such, it includes the (*Acc* $\ni$) and (*Wf base*) rules, which are never required for any proofs in the current system. Otherwise, the subobject-selection judgments can be seen as motivating the other rules.

The (*Sel class*) rule evaluates the subterm T in an environment extended to include a and $a \ni m_1, \ldots, a \ni m_n$. The well-formedness of environments ensures that a is not already defined in E. The member names $m_1, \ldots, m_n$ are assumed to be a set.

To select the primary subobject of a using (*Sel pri*), we have only to show that a is a class in E. This is shown by application of (*Acc a*), which in turn forces E to be well-formed.

Environments

$$\frac{}{\vdash \emptyset\ env}\ (Env\ \emptyset) \qquad \frac{\vdash E\ env \quad a \notin \mathrm{dom}(E)}{\vdash E, a\ env}\ (Env\ a) \qquad \frac{E \vdash a\ class}{\vdash E, a \ni m\ env}\ (Env\ \ni)$$

Access

$$\frac{\vdash E, a, E'\ env}{E, a, E' \vdash a\ class}\ (Acc\ a) \qquad \frac{\vdash E, a \ni m, E'\ env}{E, a \ni m, E' \vdash a \ni m}\ (Acc\ \ni)$$

Well-formed Subobjects

$$\frac{E \vdash a\ class}{E \vdash a/a\ wf}\ (Wf\ base)$$

Subobject Selection

$$\frac{E \vdash a\ class}{E \vdash a \rhd a/a}\ (Sel\ pri) \qquad \frac{E, a, a \ni m_1, \dots, a \ni m_n \vdash T \rhd a/c}{E \vdash \mathbf{class}(a; m_1, \dots, m_n)\ \mathbf{in}\ T \rhd a/c}\ (Sel\ class) \quad (n \geq 0)$$

2.3 CSC Basic Properties

With the CSC system we can prove such assertions as

$$\vdash \mathbf{class(A;)\ in\ A} \rhd \mathbf{A/A}$$
$$\vdash \mathbf{class(A;x,y)\ in\ class(B;z)\ in\ A} \rhd \mathbf{A/A}$$
$$\vdash \mathbf{class(A;x,y)\ in\ class(B;z)\ in\ B} \rhd \mathbf{B/B}$$

We can state some easy properties of CSC.

Proposition 2.1 (CSC Properties)
Given an E, a term T, and a query Q:

(i) *If $E \vdash Q \rhd a/b$ then $E \vdash a/b$* wf.

(ii) *There is at most one a/b such that $E \vdash T \rhd a/b$.*

Since (*Sel pri*) is the only selector, and since primary subobjects are well-formed by (*Wf base*), both (i) and (ii) follow.

3 Single Inheritance

The single inheritance system SI is now defined as an extension of the CSC system. The rules and judgment forms of SI extend those of CSC. The result is a model of a single-inheritance system much like Simula [8] or single-inheritance C++.

We consider both late-binding (dynamic) and early-binding (static) methods as well as instance-variables, which are static in the same sense as static methods. An unusual aspect of our formalism is that, rather than annotate each member as either static or dynamic in the class declarations, we annotate the reference sites, where the information is immediately useful. We assume that these annotations are consistent with a system in which the annotations at the reference sites are inferred from some information in the class declarations. Moreover, we do not distinguish between instance variables and static methods.

3.1 SI Syntax

To complement CSC's **class**, we introduce **inh** for class definitions with inheritance. We also add three new queries: **dyn** for dynamic references, **stat** for static references, and **super** for superclass references, which are also static.

SI Syntax

$T ::=$	SI term
$\textbf{class}(a; m_1, \ldots, m_n)$ **in** T	$(n \geq 0)$ class definition
$\textbf{inh}(a; b; m_1, \ldots, m_n)$ **in** T	$(n \geq 0)$ inheriting class definition
Q	query
$Q ::=$	SI query
a	primary subobject of a
$Q.\textbf{dyn}(m)$	dynamic subobject selection
$Q.\textbf{stat}(m)$	static subobject selection
$Q.\textbf{super}(m)$	superclass subobject selection

The new SI judgments extend the CSC judgments in order to express the effects of inheritance. First, $E \vdash a \prec b$ asserts that a inherits directly from b. For subobject selection, however, we must be able to compare subobjects, as in $E \vdash a/b \leq a/c$. Finally, we need a way to express the effects of subobject ordering on the search for a subobject that defines a member m; this is given by $E \vdash \text{selects}(a, m, c)$, which says that the a/c subobject of a is selected by m.

SI Judgments (extending CSC judgments)

access	
$E \vdash a \prec b$	a inherits directly from b
subobject relations	
$E \vdash a/b \leq a/c$	a/b is more defined than a/c
$E \vdash \text{selects}(a, m, c)$	m selects a's a/c subobject

3.2 SI Rules

The SI rules, listed below, are driven by the syntax-based rules. The new inheritance rule (*Sel inh sing*) now extends E with $a \prec b$, motivating (*Env* $\prec$); the condition in (*Env* $\prec$) that $a \notin \mathrm{codom}(E)$ ensures that inheritance is acyclic in well-formed environments. The remaining syntax-based rules, (*Sel dyn*), (*Sel stat*), and (*Sel sup*), are all defined in terms of the *selects* relation, (*Rel sel*). This rule is the key to the entire system. We paraphrase (*Rel sel*) as follows:

> Subobject selection for a method m in a class a is a two-step process. First we determine the set S of candidates—the set of all subobjects of a in which m is defined. Then we select a/c, which is the least element of S.

This rule fails when m is undefined in every ancestor of a (including a). It also fails when S has incomparable minima, which indicates an ambiguous reference.

Of particular interest are the different ways in which *selects* is used by (*Sel dyn*), (*Sel stat*), and (*Sel sup*). In dynamic selection, the current static context of the reference is ignored; only the actual class of the object is considered. This is reflected in (*Sel dyn*) by ignoring b, which represents the current view of an a object. Only the actual class a is used in subobject selection, yielding a/c, another subobject of a.

In contrast, static selection is calculated with respect to the current static context, disregarding the actual class of the object. Thus, (*Sel stat*) uses b for its subobject selection. This yields a subobject b/c of b. Since b is an ancestor of a, it must be that c is also. The static selection therefore yields a/c. The case of (*Sel sup*) is nearly identical to (*Sel stat*), except that the immediate base class of the current static context is used to resolve the reference. Thus, super-method invocations are also static references.

The (*Rel sel*) rule motivates the rest of the system. The notion of a well-formed subobject is explicitly called for, as is the need for (*Acc* $\ni$), which was defined in CSC. Subobjects are compared using $\leq$, which is defined by (*Rel* $\leq$ *refl*), (*Rel* $\leq$ *arc*), and (*Rel* $\leq$ *trans*). Finally, (*Acc* $\prec$) is required by (*Sel sup*), (*Rel* $\leq$ *arc*), and (*Wf ind*).

The well-formed subobject rules for SI implement the notion that a class a has one subobject a/b for each ancestor class b. The term *ancestor* is taken to denote the transitive and reflexive closure of the $\prec$ relation. A very similar thing happens in the $\leq$ rules. The $\leq$ relation is the reflexive and transitive closure of $\prec$ as if it applied to subobjects. That is, $a/b \leq a/c$ if and only if c is an ancestor of b.

Environments

(*Env* $\prec$)

$$\frac{E \vdash a\ \mathit{class} \quad E \vdash b\ \mathit{class} \quad a \neq b \quad a \notin \mathrm{codom}(E)}{\vdash E, a \prec b\ \mathit{env}}$$

Access

$(\mathit{Acc} \prec)$

$$\frac{\vdash E, a \prec b, E' \; env}{E, a \prec b, E' \vdash a \prec b}$$

Well-formed Subobjects

$(\mathit{Wf\ ind})$

$$\frac{E \vdash a/b \; wf \quad E \vdash b \prec c}{E \vdash a/c \; wf}$$

Subobject Relations

$(\mathit{Rel} \leq \mathit{refl})$

$$\frac{E \vdash a/b \; wf}{E \vdash a/b \leq a/b}$$

$(\mathit{Rel} \leq \mathit{arc})$

$$\frac{E \vdash a/b \; wf \quad E \vdash a/c \; wf \quad E \vdash b \prec c}{E \vdash a/b \leq a/c}$$

$(\mathit{Rel} \leq \mathit{trans})$

$$\frac{E \vdash a/b \leq a/c \quad E \vdash a/c \leq a/d}{E \vdash a/b \leq a/d}$$

$(\mathit{Rel\ sel})$

$$\frac{S = \{a/b \mid E \vdash a/b \; wf \quad E \vdash b \ni m\} \quad a/c \in S \quad \forall (a/d \in S)[E \vdash a/c \leq a/d]}{E \vdash \mathrm{selects}(a, m, c)}$$

Subobject Selection

$(\mathit{Sel\ inh\ sing})$

$$\frac{E, a, a \prec b, a \ni m_1, \ldots, a \ni m_n \vdash T \rhd a/c}{E \vdash \mathbf{inh}(a; b; m_1, \ldots, m_n) \; \mathbf{in} \; T \rhd a/c} \quad (n \geq 0)$$

$(\mathit{Sel\ dyn})$

$$\frac{E \vdash Q \rhd a/b \quad E \vdash \mathrm{selects}(a, m, c)}{E \vdash Q.\mathbf{dyn}(m) \rhd a/c}$$

$(\mathit{Sel\ stat})$

$$\frac{E \vdash Q \rhd a/b \quad E \vdash \mathrm{selects}(b, m, c)}{E \vdash Q.\mathbf{stat}(m) \rhd a/c}$$

$(\mathit{Sel\ sup})$

$$\frac{E \vdash Q \rhd a/b \quad E \vdash b \prec c \quad E \vdash \mathrm{selects}(c, m, d)}{E \vdash Q.\mathbf{super}(m) \rhd a/d}$$

3.3 SI Basic Properties

As a demonstration of some of the important properties of SI, let Z abbreviate the string $\mathbf{class(A; x, y)\ in\ inh(B; A; y)}$. Then we can obtain the following theorems in SI:

$\vdash Z$ **in B**	$\rhd$ **B/B**
$\vdash Z$ **in B.dyn(x)**	$\rhd$ **B/A**
$\vdash Z$ **in B.stat(x)**	$\rhd$ **B/A**
$\vdash Z$ **in B.stat(x).stat(y)**	$\rhd$ **B/A**
$\vdash Z$ **in B.stat(x).dyn(y)**	$\rhd$ **B/B**
$\vdash Z$ **in B.stat(x).dyn(y).super(y)**	$\rhd$ **B/A**

Proposition 3.1 (SI Properties)
Given an environment E, a term T, and a query Q:

(i) *If $E \vdash Q \rhd a/b$ then $E \vdash a/b$* wf.

(ii) *There is at most one a/b such that $E \vdash T \rhd a/b$.*

(iii) *If $E \vdash Q \rhd a/b$ and if Q' is a subterm of Q then there exists c such that $E \vdash Q' \rhd a/c$.*

(iv) *Suppose $E \vdash Q \rhd a/b$ and $E \vdash Q' \rhd a/c$. If $E \vdash Q.\mathbf{dyn}(m) \rhd a/d$ then $E \vdash Q'.\mathbf{dyn}(m) \rhd a/d$.*

(v) *Suppose $E \vdash Q \rhd a/b$ and $E \vdash Q' \rhd a/b$. If $E \vdash Q.\mathbf{stat}(m) \rhd a/c$ then $E \vdash Q'.\mathbf{stat}(m) \rhd a/c$.*

These are trivial except (ii). Certainly (*Sel pri*) selects at most one subobject. The other selectors, (*Sel dyn*), (*Sel stat*) and (*Sel sup*), select as many subobjects as can be proven by (*Rel sel*) with a and m fixed. Fixing a and m also fixes S, so the only question is whether there is more than one a/c such that $\forall(a/d \in S)[E \vdash a/c \leq a/d]$. Clearly, if $a/e \in S$ and $\forall(a/e \in S)[E \vdash a/e \leq a/d]$ then $E \vdash a/e \leq a/c$. But $E \vdash a/c \leq a/e$ since $a/e \in S$. Thus $a/c = a/e$ if $\leq$ is a partial order over S. Reflexivity and transitivity of $\leq$ follow from (*Rel* $\leq$ *refl*) and (*Rel* $\leq$ *trans*), respectively. Antisymmetry is proved by the following series of lemmas.

Lemma 3.2 ($\prec$ is acyclic)
There is no E, $a_1, \ldots, a_n$ $(n > 1)$ such that

(i) $\vdash E$ env,

(ii) *for $1 \leq i < n$, $E \vdash a_i \prec a_{i+1}$, and*

(iii) $E \vdash a_n \prec a_1$.

Proof. For (i)–(iii) to hold, it must be that $a_i \prec a_{i+1}$, $a_n \prec a_1$ must all occur in E. Consider the one of these that occurs rightmost in E. It cannot be $a_i \prec a_{i+1}$ because (since it is last) a_i must already occur in the codomain of E, either as $a_{i-1} \prec a_i$ (if $i > 1$), or $a_n \prec a_i$ (if $i = 1$). Similarly, it cannot be $a_n \prec a_1$, because a_n already is in codom(E), as $a_{n-1} \prec a_n$.

```
class A {
  public:
    int x() { return( 10 + this->y() ); }
    virtual int y() { return( 0 ); } };
class B : public A {
  public:
    int y() { return( 1 ); } };

int main(void) {
    B *bp = new(B);
    printf("bp->x() ==> %d\n", bp->x());
}
```

Fig. 1. C++ single-inheritance example.

Lemma 3.3
If $E \vdash a/b \leq a/c$ then $E \vdash b \prec^* c$, where $\prec^*$ is the reflexive, transitive closure of $\prec$.

Proof. By induction on the derivation of $E \vdash a/b \leq a/c$. If the last step is (*Rel* $\leq$ *refl*) or (*Rel* $\leq$ *arc*) then the conclusion is true, and the hypothesis is clearly preserved by (*Rel* $\leq$ *trans*).

Lemma 3.4
If $E \vdash a/b \leq a/c$ and $E \vdash a/c \leq a/b$ then $b = c$.

Proof. Assume b and c are distinct. By Lemma 3.3, we deduce that $E \vdash b \prec^* c$ and $E \vdash c \prec^* b$. But this contradicts Lemma 3.2.

3.4 SI Example

To demonstrate the relationship between our subobject selection calculi and the semantics of an object-oriented language, consider the following translation from C++ into a hypothetical intermediate language that is similar in syntax to SI. From there, we show how SI relates to the semantics of the resulting code fragment.

We begin with a single-inheritance C++ program, shown in Fig. 1, which prints "`bp->x() ==> 11`". We translate this code into a more suitable intermediate language, without describing the language in detail. In the following code, class definitions are nested, as in SI, to construct a class context. An instance is created within the scope of this context, and the `x` member is referenced. This reference to `x` uses `stat` because `x` is a non-virtual method in the C++ version. The `y` reference, on the other hand, uses `dyn` because `y` is virtual in the original.

```
class(A; x=fun(self) 10 + self.dyn(y),
         y=fun(self) 0 ) in
  inh(B; A; y=fun(self) 1 ) in
    let bp = new(B) in
      bp.stat(x)
```

We are not concerned with the run-time system implied by this code fragment. Rather, we focus on the subobject-selection issues. The subobjects of **B** in this system are $\{\mathbf{B/B}, \mathbf{B/A}\}$. There are three sites in the code that call for subobject selection: `new(B)`, `bp.stat(x)`, and `self.dyn(y)`. The behavior at each site is determined by the following theorems of SI, where Z abbreviates **class(A; x, y) in inh(B; A; y)**:

$$\begin{array}{ll} \vdash Z \textbf{ in B} & \rhd \mathbf{B/B} \\ \vdash Z \textbf{ in B.stat(x)} & \rhd \mathbf{B/A} \\ \vdash Z \textbf{ in B.stat(x).dyn(y)} & \rhd \mathbf{B/B} \end{array}$$

The first of these says that, in a class context that defines the class **A** with members **x** and **y**, that also defines the class **B**, inheriting from **A** and defining the member **y**, the primary subobject of **B** is **B/B**, which represents a new instance of **B**. The second says that, in this same context, if we start with a fresh instance of **B** and make a static reference to **x**, the **B/A** subobject of **B** is selected. Finally, if we start with a new instance of **B**, make a static reference to **x**, and then—from the resulting **B/A** context—make a dynamic reference to **y**, the **B/B** subobject is selected.

4 Multiple-inheritance Core

We now define a core system for multiple inheritance as an extension of CSC. Significantly, MIC does not extend SI. Although SI shares many similarities with the multiple-inheritance systems, only the CSC subset of SI can be shared without modifications.

Like CSC, MIC is meant only as a foundation for the systems that follow. It comprises the rules that are common to those systems, but is not coherent on its own. (For example, it has no syntax for class definitions.) Nevertheless, we take this opportunity to look closely at these common rules, and to deal with some of the complications that arise in the move to multiple inheritance.

The most obvious change from SI is an extended subobject notation. In multiple inheritance, it is sometimes necessary to distinguish the two versions of a subobject a/c that arise when there are two inheritance paths between a and c. Subobject notation is therefore extended to allow sequences of classnames, as in a/α. We use α, β and γ to denote (possibly empty) sequences of class names. We freely use concatenation of sequences and individual classes, as in $a/\alpha b\beta$.

Consider the subobject $a/\alpha b$. Here, a is the actual class, b is the effective class, and α is a sequence of class names. In RMI (Sect. 5), α is a full inheritance path from a to b. In SMI (Sect. 6), α is empty. In SRMI (Sect. 7), α is a subpath satisfying certain properties to be described later.

4.1 MIC Syntax

Since each of the ensuing multiple-inheritance systems defines its own **inh** syntax, MIC does not provide one. It does provide **dyn** and **stat**, which are analogous to those in SI, but there is no obvious multiple-inheritance analog for SI's **super**, since there is no longer an obvious total order among the subobjects. (The question of whether to impose a total order, and what total order to use, is still a source of discussion [9, 12], especially for the CLOS [25] and Dylan [22] communities.)

MIC Syntax

T ::=	MIC term
Q	query
Q ::=	MIC query
a	primary subobject of a
Q.**dyn**(m)	dynamic subobject selection
Q.**stat**(m)	static subobject selection

Judgments in MIC include those in CSC, and some new forms. The $\leq$ judgment here is familiar from SI, as is the explicit subobject selection relation. These simply have been extended to the multiple-inheritance notation for subobjects. An additional judgment specifies the result of an injection from c's subobjects into a's. This requires a subobject of a to disambiguate the different images of c's subobjects in a.

MIC Judgments (extending CSC judgments)

subobject relations	
$E \vdash a/\alpha \leq a/\beta$	a/α is more defined than a/β
$E \vdash \text{selects}(a, m, \alpha)$	m selects a's a/α subobject
$E \vdash \text{inj}(a/\alpha, c/\beta, a/\gamma)$	a/γ is the image of c's c/β in a, wrt a/α

4.2 MIC Rules

A number of the MIC rules are nearly copies of the corresponding rules in SI, using the extended subobject notation. These include (*Sel dyn*), (*Rel sel*), and (*Rel $\leq$ refl*). The remaining rules require further explanation.

In (*Sel stat*), subobject selection yields b/β, which we might expect to correspond to a/β, as in SI. But in SI this is an implicit injection from the space of subobjects of b to the space of subobjects of a: Suppose $a \prec b$. If b's subobjects are $\{b/c_0, \ldots, b/c_n\}$, the subobjects of a would be $\{a/a, a/c_0, \ldots, a/c_n\}$, so every b/c_i has exactly one image a/c_i in a's subobjects.

With multiple inheritance, however, it may be the case that a has two ancestors, b and c, with subobjects b/d and c/d respectively. If we use the same injection as in SI, both of these subobjects inject to the same a/d subobject of a; this is what we call ***shared*** inheritance of d, which is discussed in Sect. 6. If we want the two subobjects to map to different subobjects of a, we need a more elaborate injection. In Sect. 5, where we introduce *replicating* inheritance, the injection maps b/d to a/bd and c/d to a/cd. For now, we have only the (*Rel inj id*) rule, which is simply an identity map.

MIC is also incomplete because the notion of well-formed subobjects is incomplete, as is the notion of a more-defined subobject. The rules (*Rel* $\leq$ *repl*) and (*Rel* $\leq$ *shar*) are used in later sections to tie the $\leq$ relation to the arcs in E, much like (*Rel* $\leq$ *arc*) in SI.

Subobject Relations

(*Rel inj id*)

$$\frac{}{E \vdash \mathrm{inj}(a/\alpha, a/\beta, a/\beta)}$$

(*Rel* $\leq$ *trans*)

$$\frac{E \vdash a/\alpha \leq a/\beta \quad E \vdash a/\beta \leq a/\gamma}{E \vdash a/\alpha \leq a/\gamma}$$

(*Rel* $\leq$ *refl*)

$$\frac{E \vdash a/\alpha \; wf}{E \vdash a/\alpha \leq a/\alpha}$$

(*Rel sel*)

$$\frac{\begin{array}{c} S = \{a/\alpha b \mid E \vdash a/\alpha b \; wf \quad E \vdash b \ni m\} \\ a/\beta \in S \quad \forall (a/\gamma \in S)[E \vdash a/\beta \leq a/\gamma] \end{array}}{E \vdash \mathrm{selects}(a, m, \beta)}$$

Subobject Selection

(*Sel dyn*)

$$\frac{E \vdash Q \rhd a/\alpha \quad E \vdash \mathrm{selects}(a, m, \beta)}{E \vdash Q.\mathbf{dyn}(m) \rhd a/\beta}$$

(*Sel stat*)

$$\frac{E \vdash Q \rhd a/\alpha b \quad E \vdash \mathrm{selects}(b, m, \beta) \quad E \vdash \mathrm{inj}(a/\alpha b, b/\beta, a/\gamma)}{E \vdash Q.\mathbf{stat}(m) \rhd a/\gamma}$$

5 Replicating Multiple Inheritance

We are now ready to present our first complete multiple-inheritance system, RMI. The defining assumption in RMI is that a class has one subobject for each inheritance path to each ancestor. Subobjects in RMI encode the complete path from the actual class of the object to one of its ancestors. If that ancestor is reached by different paths, these different paths will cause the subobjects to be annotated differently, resulting in distinct subobjects. Thus, if a class **C** inherits directly from **A** and **B**, where **B** also inherits directly from **A**, objects of **C** will have two subobjects for **A**. These are annotated **C/CA** and **C/CBA**. In RMI, each subobject is actually of the form $a/a\alpha$; that is, each encoded path begins with the actual class of the object. This will not be the case in the other systems we study.

5.1 RMI Syntax

The only new syntax introduced by RMI is the **inh** form. Here, $b_1, \dots, b_k$ are multiple immediate base classes, which we may assume to be a set. The **class** syntax from CSC is dropped; we can now use an empty series of base classes to model the equivalent behavior.

RMI Syntax

$T ::=$	RMI term
$\quad \mathbf{inh}(a; b_1, \dots, b_k; m_1, \dots, m_n)\ \mathbf{in}\ T$	$(k, n \geq 0)$ class def.
$\quad Q$	query
$Q ::=$	RMI query
$\quad a$	primary subobject of a
$\quad Q.\mathbf{dyn}(m)$	dynamic selection
$\quad Q.\mathbf{stat}(m)$	static selection

Judgments in RMI extend those in MIC with one new form: $E \vdash a \prec_r b$. The $\prec_r$ relation is analogous to the $\prec$ relation in SI, except that now we are identifying this as a replicating arc.

5.2 RMI Rules

The RMI rules are surprisingly similar to their counterparts in SI. The (*Sel inh repl*) rule is a simple extension of (*Sel inh sing*), and the $\prec_r$ relation is essentially identical to SI's $\prec$. The remaining rules are defined largely in terms of concatenation of paths, which deserves some discussion.

The (*Wf repl*) rule says that a subobject is well formed if it extends a well-formed subobject's inheritance path by one class, where that class is an immediate base class of the effective class of the original subobject. Similarly, (*Rel $\leq$ repl*) says that one subobject is immediately more defined than another if the other's effective class is an immediate base class of the effective class of the original subobject.

Finally, the injection rule (*Rel inj repl*) is more subtle than it might appear. It says that injection is only well defined if the actual class of the subobject that is being injected is the effective class of the current static context (as represented by b in the rule). Given this circumstance, the injection is formed by splicing the paths αb and $b\beta$ to arrive at $\alpha b\beta$. The motivation is that αb tells how to get from

a to a particular b, and $b\beta$ tells how to get from any b to a particular ancestor of b. The spliced path uses the first path to pin down the second.

Environments

$(Env \prec_r)$

$$\frac{E \vdash a\ class \quad E \vdash b\ class \quad a \neq b \quad a \notin \mathrm{codom}(E)}{\vdash E, a \prec_r b\ env}$$

Access

$(Acc \prec_r)$

$$\frac{\vdash E, a \prec_r b, E'\ env}{E, a \prec_r b, E' \vdash a \prec_r b}$$

Well-formed Subobjects

$(Wf\ repl)$

$$\frac{E \vdash a/\alpha b\ wf \quad E \vdash b \prec_r c}{E \vdash a/\alpha bc\ wf}$$

Subobject Relations

$(Rel\ inj\ repl)$

$$\frac{}{E \vdash \mathrm{inj}(a/\alpha b, b/b\beta, a/\alpha b\beta)}$$

$(Rel \leq repl)$

$$\frac{E \vdash a/\alpha\ wf \quad E \vdash a/\alpha b\ wf}{E \vdash a/\alpha \leq a/\alpha b}$$

Subobject Selection

$(Sel\ inh\ repl)$

$$\frac{E, a, a \prec_r b_1, \ldots, a \prec_r b_k, a \ni m_1, \ldots, a \ni m_n \vdash T \rhd a/\alpha}{E \vdash \mathbf{inh}(a; b_1, \ldots, b_k; m_1, \ldots, m_n)\ \mathbf{in}\ T \rhd a/\alpha} \quad (k, n \geq 0)$$

5.3 RMI Basic Properties

As a demonstration of some of the important properties of RMI, let Z abbreviate the string $\mathbf{inh(A;;x,y)\ in\ inh(B;A;w,z)\ in\ inh(C;A,B;y,z)}$. Then we can obtain the following theorems in RMI:

$$\begin{array}{ll}
\vdash Z\ \mathbf{in\ C} & \rhd \mathbf{C/C} \\
\vdash Z\ \mathbf{in\ C.stat(w)} & \rhd \mathbf{C/CB} \\
\vdash Z\ \mathbf{in\ C.stat(w).stat(y)} & \rhd \mathbf{C/CBA} \\
\vdash Z\ \mathbf{in\ C.stat(w).stat(y).dyn(y)} & \rhd \mathbf{C/C} \\
\vdash Z\ \mathbf{in\ C.stat(w).stat(y).dyn(z)} & \rhd \mathbf{C/C}
\end{array}$$

There is no a/α, however, such that $\vdash Z$ **in** C.**stat**(x) $\rhd a/\alpha$, even though **x** is defined in an ancestor of **C**. This is because the set S in (*Rel sel*) is $\{\mathbf{C/CA}, \mathbf{C/CBA}\}$, and these two subobjects are incomparable by $\leq$. Such a reference is said to be *ambiguous.*

Proposition 5.1 (RMI Properties)
Given an environment E, a term T, and a query Q:

(i) *If $E \vdash Q \rhd a/\alpha$ then $E \vdash a/\alpha$* wf.

(ii) *There is at most one a/α such that $E \vdash T \rhd a/\alpha$.*

(iii) *If $E \vdash Q \rhd a/\alpha$ and if Q' is a subterm of Q then there exists β such that $E \vdash Q' \rhd a/\beta$.*

(iv) *Suppose $E \vdash Q \rhd a/\alpha$ and $E \vdash Q' \rhd a/\beta$. If $E \vdash Q.\mathbf{dyn}(m) \rhd a/\gamma$ then $E \vdash Q'.\mathbf{dyn}(m) \rhd a/\gamma$.*

(v) *Suppose $E \vdash Q \rhd a/\alpha$ and $E \vdash Q' \rhd a/\alpha$. If $E \vdash Q.\mathbf{stat}(m) \rhd a/\beta$ then $E \vdash Q'.\mathbf{stat}(m) \rhd a/\beta$.*

Again, (ii) is the only non-trivial point. The proof of (ii) is essentially identical to the proof for Proposition 3.1. Antisymmetry of $\leq$ follows from (*Rel* $\leq$ *repl*), which relies on the well-formedness of subobjects (*Wf repl*) to tie the subobject orderings to the order $\prec_r$ of the class hierarchy.

6 Shared Multiple Inheritance

Our second complete multiple-inheritance system, SMI, closely parallels RMI, but differs in most details. The defining assumption of SMI is that each ancestor class leads to exactly one subobject in an inheriting class, regardless of the number of paths by which it is inherited. This is, in fact, very similar to the situation in SI, where a subobject is uniquely named by the actual class and the effective class. It might be surprising, then, to find sequences and concatenation in the semantics of SMI. The reason is that we define SMI with extra generality so that later it can be merged with RMI without redefinition. For the present, it is consistent to interpret every subobject of the form $a/\alpha b$ as equivalent to a/b.

6.1 SMI Syntax

The syntax of SMI is identical to that of RMI.

SMI Syntax

$T ::=$	SMI term
$\quad \mathbf{inh}(a; b_1, \ldots, b_k; m_1, \ldots, m_n)$ **in** T	$\quad (k, n \geq 0)$ class def.
$\quad Q$	$\quad$ query
$Q ::=$	SMI query
$\quad a$	$\quad$ primary subobject of a
$\quad Q.\mathbf{dyn}(m)$	$\quad$ dynamic selection
$\quad Q.\mathbf{stat}(m)$	$\quad$ static selection

As in RMI, judgments in SMI extend those of MIC with one new form: $E \vdash a \prec_s b$, in this case. The only difference from RMI is that $\prec_s$ is introduced in place of $\prec_r$, indicating a shared inheritance arc.

6.2 SMI Rules

The only significant differences between RMI and SMI lie in the (*Wf shar*), (*Sel inh shar*), and (*Rel* $\leq$ *shar*) rules. A subobject a/c is well formed in SMI if either $a = b$ (*Wf base*), or there exists another well-formed subobject such that c is an immediate base class of its effective class. The only reason the (*Wf shar*) rule uses $a/\alpha b$ rather than simply a/b is so that it still applies when this system is combined with RMI; in SMI, α is always empty, so only b is significant. This establishes that every well-formed subobject in SMI has only a singleton sequence in its notation. If we simplify (*Rel* $\leq$ *shar*) similarly, it is clearly the same ordering we described for SI. Finally, (*Rel inj shar*) is exactly the injection that is implicit in SI's (*Rel inj sing*).

Environments

(*Env* $\prec_s$)

$$\frac{E \vdash a \ \mathit{class} \quad E \vdash b \ \mathit{class} \quad a \neq b \quad a \notin \mathrm{codom}(E)}{\vdash E, a \prec_s b \ \mathit{env}}$$

Access

(*Acc* $\prec_s$)

$$\frac{\vdash E, a \prec_s b, E' \ \mathit{env}}{E, a \prec_s b, E' \vdash a \prec_s b}$$

Well-formed Subobjects

(*Wf shar*)

$$\frac{E \vdash a/\alpha b \ \mathit{wf} \quad E \vdash b \prec_s c}{E \vdash a/c \ \mathit{wf}}$$

Subobject Relations

(*Rel inj shar*)

$$\frac{}{E \vdash \mathrm{inj}(a/\alpha b, b/c, a/c)}$$

(*Rel* $\leq$ *shar*)

$$\frac{E \vdash a/\alpha b \; wf \quad E \vdash a/c \; wf \quad E \vdash b \prec_s c}{E \vdash a/\alpha b \leq a/c}$$

Subobject Selection

(*Sel inh shar*)

$$\frac{E, a, a \prec_s b_1, \ldots, a \prec_s b_k, a \ni m_1, \ldots, a \ni m_n \vdash T \rhd a/\beta}{E \vdash \mathbf{inh}(a; b_1, \ldots, b_k; m_1, \ldots, m_n) \; \mathbf{in} \; T \rhd a/\beta} \qquad (k, n \geq 0)$$

6.3 SMI Basic Properties

As a demonstration of some of the important properties of SMI, let Z abbreviate the string $\mathbf{inh(A;;x,y)\ in\ inh(B;A;x,z)\ in\ inh(C;A;y,z)\ in\ inh(D;B,C;)}$. Then we can obtain the following theorems in SMI:

$$\begin{array}{ll}
\vdash Z \ \mathbf{in\ D} & \rhd \mathbf{D/D} \\
\vdash Z \ \mathbf{in\ D.stat(x)} & \rhd \mathbf{D/B} \\
\vdash Z \ \mathbf{in\ D.stat(x).stat(z)} & \rhd \mathbf{D/B} \\
\vdash Z \ \mathbf{in\ D.stat(x).stat(y)} & \rhd \mathbf{D/A} \\
\vdash Z \ \mathbf{in\ D.stat(x).dyn(y)} & \rhd \mathbf{D/C} \\
\vdash Z \ \mathbf{in\ D.stat(y)} & \rhd \mathbf{D/C} \\
\vdash Z \ \mathbf{in\ D.stat(y).stat(z)} & \rhd \mathbf{D/C} \\
\vdash Z \ \mathbf{in\ D.stat(y).stat(x)} & \rhd \mathbf{D/A} \\
\vdash Z \ \mathbf{in\ D.stat(y).dyn(x)} & \rhd \mathbf{D/B}
\end{array}$$

There is no a/α, however, such that $\vdash Z$ **in D.stat(y).dyn(z)** $\rhd a/\alpha$, even though **z** is defined in two ancestors of **D**. The reference is ambiguous because the set $\{\mathbf{D/B}, \mathbf{D/C}\}$, which is the set S in (*Rel sel*), has no least element.

Proposition 6.1 (SMI Properties)
The properties listed in Proposition 5.1 *hold for SMI also.*

As in SI, antisymmetry follows directly from (*Rel* $\leq$ *shar*), which bases subobject orderings on the acyclic order $\prec_s$ over class names.

7 Combined Shared and Replicating Multiple Inheritance

Finally, we may now develop the complete, merged multiple-inheritance system, SRMI. As with the other multiple-inheritance systems, SRMI uses the extended notation for subobjects. The semantics is almost entirely defined by the two systems RMI and SMI, with only two new rules added for SRMI. This merged system is essentially the same system we have previously defined with a less-formal semantics [17]. There, we demonstrated that this system captures the key semantic issues that arise in the C++ multiple-inheritance model.

7.1 SRMI Syntax

One new syntactic form is introduced—a variation on the **inh** form in which two sequences of base classes may be specified. The first sequence is interpreted as shared base classes, the second as replicating bases.

SRMI Syntax

$T ::=$	SRMI term
$\textbf{inh}(a; b_1, \ldots, b_k; c_1, \ldots, c_l; m_1, \ldots, m_n)$ **in** T	$(k, l, n \geq 0)$
Q	query
$Q ::=$	SRMI query
a	primary
$Q.\textbf{dyn}(m)$	dynamic
$Q.\textbf{stat}(m)$	static

Judgments in SRMI are simply the combined judgments of RMI and SMI.

7.2 SRMI Rules

Only two rules are added to form SRMI, but three rules are also removed. The new (*Sel inh comb*) rule is a simple extension of the analogous rules in RMI and SMI. The new (*Rel inj comb*) rule is very similar to SMI's (*Rel inj shar*), but includes a new condition. This condition, that $b \neq c$, ensures that c is a shared base class of b; the same condition would have meant something quite different (and undesirable) if it were included in the SMI rule. In fact, many of the interactions between the RMI and SMI versions of related rules deserve special attention.

Take, for example, the specification of well-formed subobjects. A well-formed subobject in SRMI is a subobject $a/c_1, \ldots, c_n$, $(n \geq 1)$ in which either $c_1 = a$ or there exists an ancestor b of a such that $b \prec_s c_1$. Moreover, $c_i \prec_r c_{i+1}$, for $0 \leq i < n$. In SRMI, this relationship is given a concise, formal definition in the three rules (*Wf base*), (*Wf repl*), and (*Wf shar*). Thus, the aggregation of rules in SRMI leads to a rich interaction of the two systems on which it is based.

Proposition 7.1
The properties listed in Proposition 5.1 *hold for SRMI also.*

Again, antisymmetry for $\leq$ is the only difficult point. By the use of *codom* in (*Env* $\prec_r$) and (*Env* $\prec_s$), the combined class hierarchy is still acyclic. It is therefore not possible for the combination of (*Rel* $\leq$ *repl*) and (*Rel* $\leq$ *shar*) to violate antisymmetry.

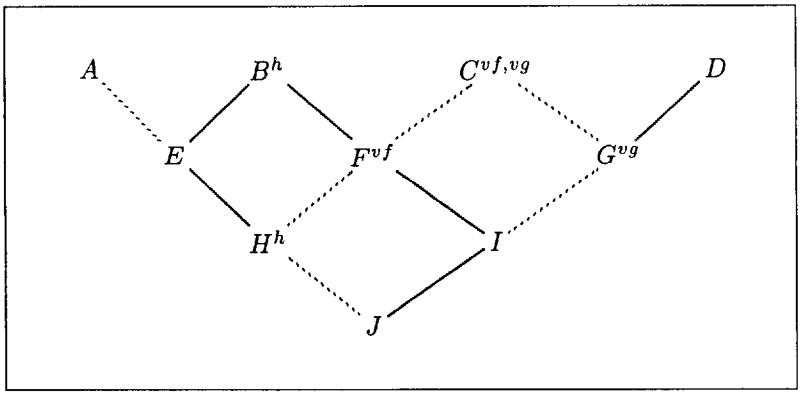

Fig. 2. A sample multiple-inheritance hierarchy. Dotted lines are shared arcs, solid lines are replicating arcs. Superscripts denote member functions of the class.

SRMI: Shared and Replicating Multiple Inheritance
(extends SMI, excluding (Sel inh shar) and (Rel inj shar))
(extends RMI, excluding (Sel inh repl))

Subobject Relations

(*Rel inj comb*)

$$\frac{b \neq c}{E \vdash \mathrm{inj}(a/\alpha b, b/c\beta, a/c\beta)}$$

Subobject Selection

(*Sel inh comb*)

$$\frac{\begin{array}{c} E, a, a \prec_s b_1, \ldots, a \prec_s b_k, \\ a \prec_r c_1, \ldots, a \prec_r c_l, \\ a \ni m_1, \ldots, a \ni m_n \\ \vdash T \rhd a/\alpha \end{array}}{\begin{array}{c} E \vdash \mathbf{inh}(a; b_1, \ldots, b_k; c_1, \ldots, c_l; m_1, \ldots, m_n) \\ \mathbf{in}\ T \rhd a/\alpha \end{array}} \quad (k, l, n \geq 0)$$

7.3 SRMI Example

In C++, members—rather than references—are divided into virtual and non-virtual. This is modeled in SRMI by agreeing that virtual members should only

```
inh(A;; ax=fun(self) 10 ) in
 inh(B;; bx=fun(self) 11,
         h=fun(self y) y + self.stat(bx)) in
  inh(C;; cx=fun(self) 12,
         vf=fun(self) fun(y) y + self.stat(cx),
         vg=fun(self)
             fun(y) self.dyn(vf)(y + self.stat(cx))) in
   inh(D;; dx=fun(self) 13 ) in
    inh(E; A; B; ex=fun(self) 100 ) in
     inh(F; C; B; fx=fun(self) 200,
                  vf=fun(self)
                      fun(y) self.stat(h)(y + self.stat(fx))) in
      inh(G; C; D; gx=fun(self) 300,
                   vg=fun(self)
                       fun(y)
                        self.dyn(vf)(y + self.stat(gx))) in
       inh(H; F; E; hx=fun(self) 1000,
                    h=fun(self)
                        fun(y) y + self.stat(hx)) in
        inh(I; G; F; ix=fun(self) 2000 ) in
         inh(J; H; I; jx=fun(self) 5000 ) in
          let ip = new(I),
              jp = new(J) in
            list(jp.dyn(vf), jp.dyn(vg), ip.dyn(vg))
```

Fig. 3. Translation of Fig. 4 into pseudo-code.

be used in **dyn** forms, with all others used only in **stat** forms. In SRMI and in C++, a virtual member function **vf** may be ambiguously referenced by an instance of class **J**, given the hierarchy in Fig. 2. In SRMI, instances of such a class will always find **vf** to be ambiguous, regardless of the static context of the dynamic reference. Unfortunately, only in the most recent semantics of C++ [1] can any similar requirement be found. This change in the C++ specification deserves further discussion.

The problem with the original C++ specification [10, 26] is that ambiguity was not considered a problem with the class definition, but rather a problem with the specific reference. Classes were allowed to inherit members ambiguously; only a flagrant reference to the ambiguous member could cause a compile-time error. Suppose, as in Fig. 2, that **J** inherits **vf** ambiguously. A reference `jp->vf()` would be clearly ambiguous, where `jp` is a pointer to an instance of **J**. This kind of reference is easily detected at compile time, and leads to the expected error message. What the original specification seems to have overlooked, however, is that the instance may be used in a static context of one of its base classes in which **vf** was not ambiguous, such as in **G**. That context was compiled without error; it assumed tacitly that any derived class would have an unambiguous definition of **vf**. We see that this was not a safe assumption. The result of this design flaw

```
class A { public: int ax; A(): ax(10) {} };
class B {
 public:
   int bx;
   B(): bx(11) {}
   int h(int y){ return( bx + y ); } };
class C {
 public:
   int cx;
   C(): cx(12) {}
   virtual int vf(int y){ return( cx + y ); }
   virtual int vg(int y){ return( this->vf( cx + y ) ); } };
class D { public: int dx; D(): dx(13) {} };
class E: public virtual A, public B {
 public: int ex; E(): ex(100) {} };
class F: public B, public virtual C {
 public:
   int fx;
   F(): fx(200) {}
   virtual int vf(int y){ return( this->h( fx + y ) ); } };
class G: public virtual C, public D {
 public:
   int gx;
   G(): gx(300) {}
   virtual int vg(int y){ return( this->vf( gx + y ) ); } };
class H: public E, public virtual F {
 public:
   int hx;
   H(): hx(1000) {}
   int h(int y){ return( hx + y ); } };
class I: public F, public virtual G {
 public: int ix; I(): ix(1000) {} };
class J: public virtual H, public I {
 public: int jx; J(): jx(50000) {} };

int main(){
 G *gp = new(G);
 I *ip = new(I);
 J *jp = new(J);
//  printf("jp->vf(0) = %d\n", jp->vf(0)); // obvious vf ambiguity
 printf("jp->vg(0) = %d\n", jp->vg(0));  // vf not ambiguous?! prints 511
 printf("ip->vg(0) = %d\n", ip->vg(0));  // prints 511
 printf("gp->vg(0) = %d\n", gp->vg(0));  // prints 312
}
```

Fig. 4. C++ multiple-inheritance example.

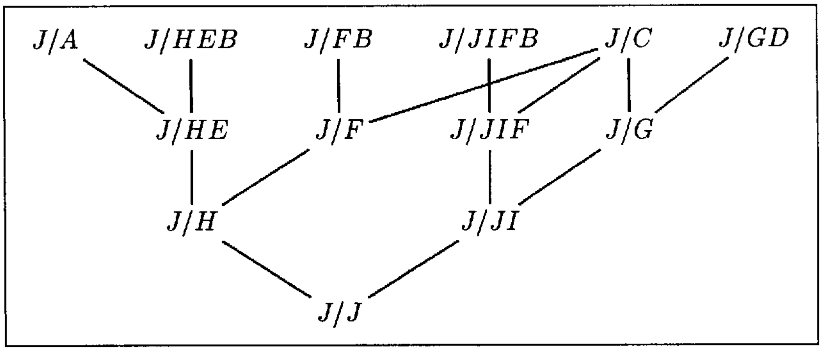

Fig. 5. The subobject poset for J in the class hierarchy shown in Fig. 2.

is that the reference that occurs in **G**'s static context actually chooses, by some hidden algorithm, one of the inherited definitions of **vf** and calls that code. Thus, **J**.**dyn**(**vf**) is not necessarily the same as **J**.**stat**(**vg**).**dyn**(**vf**), and the semantics of the latter case is underspecified.

In the recent drafts of the proposed standard for C++ [1], this problem is remedied by forcing a class to provide its own definition of any such ambiguously inherited member function. The error is now associated with the class definition rather than the references. This leads to the behavior specified in SRMI, in which a virtual member leads to the same result whatever the static context of the reference. The irony of this result is that SRMI detects ambiguity on a reference-by-reference basis, much as in the original C++ specification. In fact, C++ implementors have always had enough information to know they were getting into trouble with virtuals in base-class contexts, but their hands were tied by the earlier specification. Because they could not be certain that an instance of **J** would make its way to an inherited context that referenced **vf**, they could not give an error. The new specification allows (in fact, requires) the implementor to make the more conservative choice.

Figure 3 shows a program in a hypothetical language, much as we used in the SI example. This program corresponds to the C++ code in Fig. 4, and implements the hierarchy in Fig. 2.

SRMI agrees, with both the old and the new C++ specifications, in that `jp.dyn(vf)` is ambiguous. We notice that `jp.dyn(vg)` is also ambiguous, since it leads to `self.dyn(vf)`, where `self` is the `J` instance. As shown in Fig. 5, which contains the subobjects for `J` and their ordering, the subobjects `J/F` and `J/JIF` are unrelated. The new C++ specification says that the `J` definition should not compile at all, due to the ambiguity of `vf`. We simply show that a dynamic reference to the `vf` member of a `J` instance is equally ambiguous in any context. Finally, SRMI agrees with both the old and new specifications in that `ip.dyn(vg)` yields `312`. This is the result of calling, in turn, `G::vg`, `F::vf`, and `B::h`.

Formally, we show these three results by the proofs of the following assertions. Let Z be the appropriate string **inh(A; ; ax) in** ... **inh(J; H; I; jx)**, encoding the class hierarchy above.

There is no a/α such that

$$\begin{array}{lll} & \vdash Z \textbf{ in J.dyn(vf)} & \rhd a/\alpha \\ \text{or} & \vdash Z \textbf{ in J.stat(vg).dyn(vf)} & \rhd a/\alpha \end{array}$$

however,

$$\begin{array}{ll} \vdash Z \textbf{ inI.dyn(vg)} & \rhd \textbf{I/G} \\ \vdash Z \textbf{ inI.dyn(vg).dyn(vf))} & \rhd \textbf{I/IF} \\ \vdash Z \textbf{ inI.dyn(vg).dyn(vf).stat(h)} & \rhd \textbf{I/IFB} \end{array}$$

8 Related Work

The models of inheritance developed by Kamin [11], Reddy [16], and Cook and Palsberg [7, 6] laid the foundations for formal models of inheritance. While some of these mention the desire to model multiple inheritance, there is no comprehensive model proposed. Multiple inheritance introduces many design issues that have not been given a fully satisfactory taxonomy, although Carnese [3], Snyder [23] and Knudsen [13], for example, have made progress in this direction. Sakkinen [18] gives a comprehensive, informal introduction to the design issues surrounding subobjects in inheritance systems. Carré and Geib's point-of-view notion for multiple inheritance [4] is also aimed at understanding subobjects.

The C++ multiple-inheritance system [26] is a combination of design ideas, originating with Krogdahl's multiple-inheritance design [14]. The resulting system, as Sakkinen notes [19, 20], is best understood in terms of the subobjects of each kind of object. Snyder's informal model [24] of the C++ system, however, intentionally simplifies some of the complicating features of the multiple-inheritance system. Although Wallace [27] and Seligman [21] have developed formal models of C++, the former sheds little light on issues such as subobjects, subobject selection, and ambiguity analysis, while the latter models only single inheritance.

Unlike these models, our formalisms are not full language semantics. We develop instead a framework for resolving crucial static issues, which correspond to essential tasks that take place at compile time. Given the information obtained at this stage, a member reference in the run-time system is resolved with only a small run-time cost. We have previously presented a subobject-based algebra [17] for resolving similar issues in a system that is essentially the same as the system SRMI developed here. The current treatment, in terms of a logic system, allows us to make detailed comparisons of the semantics of a family of related inheritance systems.

Despite our interest in static issues, we do not develop a type theory for our object models. Some semantics have dealt chiefly with the type-safety vs.

expressiveness issues that arise in object-oriented languages, including those that support multiple inheritance [2, 15, 5]. The kind of multiple inheritance modeled in these systems does not resemble SRMI as much as it does SMI, which is simpler in many respects. In fact, a provably-safe static type system for a system such as SRMI is an open problem.

9 Conclusions

Expanding on our algebraic semantics of subobjects, we have presented a relatively simple calculus to describe subobjects and subobject selection explicitly. This gives us a framework for resolving crucial static issues in a formal proof system. The generality of this new methodology allows us to express a number of important design choices in inheritance-based languages, such as the distinction between a replicating and shared inheritance. This is demonstrated by the incremental development of several related inheritance systems, culminating in the system SRMI, whose semantics is a subtle combination of the preceding systems. This last system is of particular interest, due to its correspondence to the C++ inheritance model, which has no satisfactory formal characterization.

A complete picture of multiple inheritance must deal with a number of runtime issues we have omitted here, as well as an analysis of the type-safety issues, among other things. We have chosen here to focus on a simple descriptive account of the issues relating to subobject selection. Although this is a static characterization based on class names, there is no reason it cannot be applied to dynamic classes, as long as each class is somehow uniquely identified. This enriched exposition of subobjects can be seen as the foundation for a broad formal theory of multiple inheritance.

10 Acknowledgments

We would like to thank Michael Ashley, Markku Sakkinen, and the anonymous reviewers for their comments on drafts of this paper.

References

1. Information Processing Systems Accredited Standards Committee X3. Working paper for draft proposed international standard for information systems—programming language C++. Draft of 28 April 1995.

2. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

3. Daniel J. Carnese. Multiple inheritance in contemporary programming languages. Technical Report MIT/LCS/TR-328, M.I.T., Sept. 1984.

4. Bernard Carré and Jean-Marc Geib. The point of view notion for multiple inheritance. In *Proceedings OOPSLA-ECOOP '90, ACM SIGPLAN Notices*, pages 312–321, 1990.

5. Adriana B. Compagnoni and Benjamin C. Pierce. Multiple inheritance via intersection types. Technical Report ECS-LFCS-93-275, University of Edinburgh, 1993. Also Technical Report 93-18, C.S. Department, Catholic University Nijmegen.

6. William R. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989. Technical Report CS-89-33.

7. William R. Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 433–443, 1989.

8. Ole-Johan Dahl and Kristen Nygaard. Simula—an Algol-based simulation language. *CACM*, 9(9):671–678, September 1966.

9. Roland Ducournau, Michel Habib, Marianne Huchard, and Marie-Laure Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proceedings OOPSLA '94, ACM SIGPLAN Notices*, pages 164–175, 1994.

10. Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

11. Samuel Kamin. Inheritance in SMALLTALK-80: A denotational definition. In *Proceedings POPL '88*, pages 80–87, 1988.

12. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

13. Jørgen Lindskov Knudsen. Name collision in multiple classification hierarchies. In *Proceedings ECOOP '88*, LNCS 322, pages 93–108. Springer-Verlag, 1988.

14. Stein Krogdahl. Multiple inheritance in Simula-like languages. *BIT*, 25:318–326, 1984.

15. Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, December 1991.

16. Uday Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Conf. on LISP and Functional Programming*, 1988.

17. Jonathan G. Rossie Jr. and Daniel P. Friedman. An algebraic semantics of subobjects. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, pages 187–199, 1995. Published as Proceedings OOPSLA '95, ACM SIGPLAN Notices, volume 30, number 10.

18. Markku Sakkinen. Disciplined inheritance. In *Proceedings ECOOP '89*, The British Computer Society Workshop Series, pages 39–56. Cambridge University Press, 1989.

19. Markku Sakkinen. A critique of the inheritance principles of C++. *Computing Systems*, 5(1):69–110, 1992.

20. Markku Sakkinen. A critique of the inheritance principles of C++: Corrigendum. *Computing Systems*, 5(3), 1992.

21. Adam Seligman. FACTS: A formal analysis of C++: Type rules and semantics. B.A. Honors Thesis, Williams College, May 1995.

22. Andrew Shalit, Orca Starbuck, and David Moon. *Dylan Reference Manual.* Apple Computer, Inc., 1995. Draft of September 29, 1995.

23. Alan Snyder. Inheritance and the development of encapsulated software components. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. MIT Press, 1987.

24. Alan Snyder. Modeling the C++ object model, an application of an abstract object model. In *Proceedings ECOOP '91*, LNCS 512, pages 1–20. Springer-Verlag, 1991.

25. Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, 2nd edition, 1990.

26. Bjarne Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4), 1989.

27. Charles Wallace. The semantics of the C++ programming language. In Egon Boerger, editor, *Specification and Validation Methods for Programming Languages*, pages 131–163. Clarendon Press, Oxford, 1995.