A Member Lookup Algorithm for C++

G. Ramalingam and Harini Srinivasan
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY, 10598, USA
{rama, harini}@watson.ibm.com

Abstract

The member lookup problem in C++ is the problem of resolving a specified member name in the context of a specified class. Member lookup in C++ is complicated by the presence of virtual inheritance and multiple inheritance. In this paper, we present an efficient algorithm for member lookup in C++. We also present a formalism for the multiple inheritance mechanism of C++, which we use as the basis for deriving our algorithm. The formalism may also be of use as a formal basis for deriving other C++ compiler algorithms.

1 Introduction

This paper concerns member lookup in C++. When a class member access expression such as `x.m` is statically analyzed, e.g. by a compiler, the member name `m` has to be resolved in the context of a class specified by the static type of `x`. In the presence of just single inheritance, member lookup is essentially like name lookup in the presence of nested scopes (eg., as in languages like Pascal that allow nested procedures), which is fairly simple. Member lookup in C++, however, is complicated by the presence of multiple inheritance, virtual and non-virtual inheritance, and the C++ rule of dominance.

An object (i.e. , a class instance) may contain, thanks to inheritance, multiple members with the same name. The C++ dominance rule defines which members of an object *dominate* (i.e., "hide") which other members with the same name. The lookup for a member name `m` in the context of a class `D` unambiguously resolves to a particular member named `m` iff that member dominates all other members named `m` in an object of class `D`. Otherwise, the lookup is said to be ambiguous.

To appear at the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation, Las Vegas, Nevada, 15-18 June, 1997.

This seems simple enough. However, some subtleties creep in when one attempts to formalize the above explanation. As evidence of this complexity, consider the fact that the following description of the concept in the *The Annotated C++ Reference Manual* is, in fact, imprecise:

> A name `B::f` *dominates* a name `A::f` if its class `B` has `A` as a base. If a name dominates another no ambiguity exists between the two; the dominant name is used when there is a choice [5, page 204–205].

The recent draft definition of ANSI C++ makes this explanation more precise:

> A member name `f` in one subobject `B` dominates a member name `f` in subobject `A` if `A` is a base class subobject of `B` [1, Section 10.2].

The complexity arises from the difference in the semantics of virtual and non-virtual inheritance, which makes it somewhat difficult to even identify the precise set of members contained in an object!

Figures 1 and 2 illustrate the problem through two examples. Figures 1(a) and 2(a) depict two programs. The only difference between these two programs is that the first one uses non-virtual inheritance while the second uses virtual inheritance. The class hierarchy of a program may be depicted by a graph, the *class hierarchy graph*, whose nodes denote classes and whose edges denote inheritance relations. The class hierarchy graph of the programs in Figures 1(a) and 2(a) are presented in Figures 1(b) and 2(b) respectively. (The solid edges denote non-virtual inheritance, while dashed edges denote virtual inheritance.)

It turns out the lookup `p->m` is ambiguous in Figure 1(a) but not in Figure 2(a) (even though the quotation from *The Annotated C++ Reference Manual* above seems to suggest that the lookup is unambiguous in both cases). The precise details are irrelevant here, but the ultimate source of the problem is that an `E` object has two subobjects of class A in the first case, but only one subobject of class A in the second case. The trouble is that the class hierarchy graph does not make this obvious.

A different graph-based representation of the class hierarchy, called the *subobject graph*, does depict the "composition" of objects of various classes better. The subobject graph facilitates formalizing the concepts of dominance and the lookup operation. Rossie and Friedman [9] present a formalism that models the multiple inheritance mechanism of C++, where they provide formal definitions of the subobject graph, and utilize it to provide a definition of the member lookup operation. A precise description of the subobject graph can be found in [9]. Informally, the subobject graph can be generated from the class hierarchy graph by duplicating nodes as indicated by the C++ semantics of non-virtual inheritance. The subobject graphs of the examples in Figures 1(a) and 2(a) are presented in Figures 1(c) and 2(c) respectively.

Rossie and Friedman's goal is to provide a *model* of member lookup in C++, not an algorithm. (Their specification of the lookup operation, being *executable*, is itself an algorithm. However, it is a potentially inefficient one since the subobject graph's size can be exponential in the size of the class hierarchy graph.)

In this paper, we present a formalism *based on the class hierarchy graph* that models the multiple inheritance mechanism of C++. There is a close relationship between our formalism and that of Rossie and Friedman, as will be explained in the paper. However, since our formalism is based on the class hierarchy graph, it enables us to derive an efficient, polynomial time algorithm for member lookup. (The worst-case complexity of doing a single lookup using our algorithm is $O(|N| \times (|N| + |E|))$, where $|N|$ denotes the number of classes in the program, and $|E|$ denotes the number of inheritance edges in the class hierarchy graph. The complexity of a single lookup, however, reduces to $O(|N| + |E|)$ in the common case of a program with no ambiguous lookups. All possible member lookups can be done in time $O(|M| \times |N| \times (|N| + |E|))$ (or $O((|M| + |N|) \times (|N| + |E|))$ for a program with no ambiguous lookups), where $|M|$ denotes the number of member names in the program.

The primary contribution of this paper is an algorithm for member lookup in C++. (We are not aware of any previously published algorithm for this problem.) A secondary contribution of this paper is a reformulation of the Rossie-Friedman multiple inheritance formalism, which we feel may be of use as a formal basis for deriving efficient implementations of various aspects of a C++ compiler, as illustrated by this paper in the case of member lookup.

Apart from its applications in a C++ compiler (in performing static analysis and in constructing virtual-function tables), our lookup algorithm is also useful in efficiently implementing class hierarchy slicing [12].

Organization: Section 2 introduces the terminology we use. Section 3 presents our formalism. Section 4 describes the ideas behind our algorithm. Section 5 presents the algorithm and an analysis of its complexity. Section 6 discusses a few other issues relating to lookup in C++. Section 7 discusses related work and Section 8 concludes our paper.

2 Terminology

The class hierarchy of a C++ program can be compactly modelled by a directed acyclic graph, the *Class Hierarchy Graph* (CHG) whose nodes denote classes and whose edges denote inheritance relations. The CHG of a program is defined as follows.

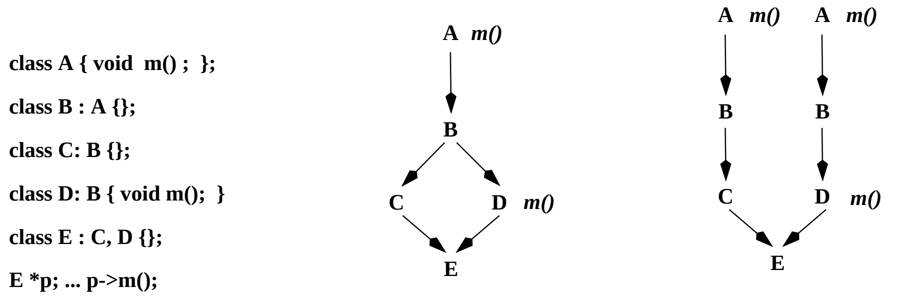
Let N denote the set of classes in the C++ program. Let E_v denote the set of all pairs of classes (X, Y) such that X is a direct virtual base of Y. Let E_{nv} denote the set of all (X, Y) such that X is a direct non-virtual base of Y. We will refer to the elements of E_v and E_{nv} as virtual and non-virtual edges respectively. Let E denote $E_v \cup E_{nv}$. We will use the notation $X \to Y$ to refer to an element (X, Y) of E. The class hierarchy graph is the tuple $\langle N, E \rangle$.

A class X is said to be a base class of another class Y iff there is a nonempty path from X to Y in the CHG. Further, X is said to be a virtual base class of Y iff there is a path from X to Y whose first edge is a virtual edge. We will denote a path in the CHG by the sequence of nodes in the path, though, for the sake of clarity, we will often denote a path consisting of a single edge as $C \to E$ rather than CE. Let α and β be two paths such that the last node of α and the first node of β are the same. We will denote the path obtained by concatenating α And β by $\alpha \bullet \beta$. For example, $(ABC) \bullet (CED)$ is $ABCED$. We say α is a prefix of $\alpha \bullet \beta$ and that β is a suffix of $\alpha \bullet \beta$. (A path is both a prefix and suffix of itself.)

We will denote the set of all paths in a graph G by $Paths(G)$.

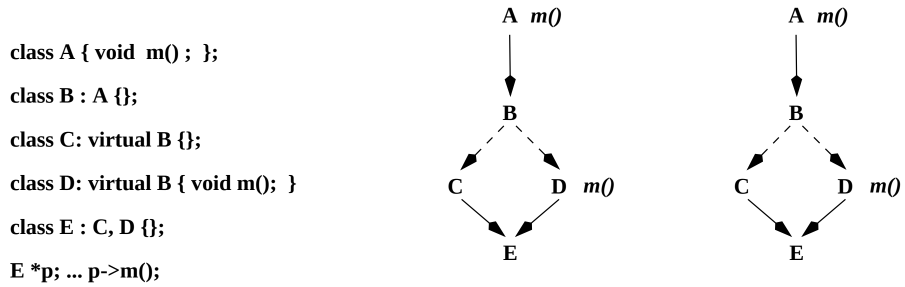
Every class X has an associated set of members, which we denote by $M[X]$, which is the set of members declared directly in that class. (In figures depicting class hierarchy graphs, elements of $M[X]$ are shown enumerated adjacent to node X.) A class, however, *inherits* all members of its base classes as well. A precise definition of the set of members that constitute a *class instance* (also called *object*) is complicated, and we will defer that to the next section.

We will not distinguish between virtual and non-



(a) Non-virtual Inheritance Example *(b) Class Hierarchy Graph* *(c) Subobject Graph*

Figure 1: An example illustrating non-virtual inheritance.



(a) Virtual Inheritance Example *(b) Class Hierarchy Graph* *(c) Subobject Graph*

Figure 2: An example illustrating virtual inheritance.

virtual members, as the distinction makes no difference for the problem we address. C++ also makes a distinction between static and non-static members, which *is* relevant to lookup. We will postpone the discussion of static members until Section 6 and assume that all members are non-static for now.

3 A Model of C++ Multiple Inheritance

The essential meaning of an inheritance edge $B \rightarrow D$ in the class hierarchy graph is that every D object contains a B object. In particular, a D object inherits all members of B as well.

If A_n is a base class of A_1, then there exists some path $A_n A_{n-1} \cdots A_2 A_1$. Such a path implies that every A_1 object contains a A_2 subobject that contains a A_3 subobject that, continuing in the same vein, eventually contains a A_n subobject.

The presence of multiple inheritance implies that multiple paths can exist between a given pair of classes B and D. Each of these paths identifies a B subobject within a D subobject. Because of the presence of both virtual and non-virtual inheritance, different paths between the same pair of classes may or may not identify the same subobject within a given object. In the *absence* of non-virtual inheritance, all paths between the same pair of classes identify the same subobject. In the *absence* of virtual inheritance, different paths always identify different subobjects.

Any formalism for explaining the multiple inheritance mechanism of C++ must begin by answering the question: what is the set of subobjects that comprises an instance of a given class? [9]. Rossie and Friedman do so by constructing a subobject graph. In this section we present a formalism similar to that of Rossie and Friedman, except that it is based on the class hierarchy graph rather than the subobject graph.

The Composition of Class Instances

We will begin by answering the question: when do two different paths in the CHG identify the same subobject?

Definition 1 *For any path α, let ldc(α) denote the source of the path, and let mdc(α) denote the target of the path.*

(Here, *ldc* stands for the *least derived class,* while *mdc* stands for the *most derived class.*)

Definition 2 *For any path α, let fixed(α) denote the longest prefix of the path that does not contain any virtual edge.*

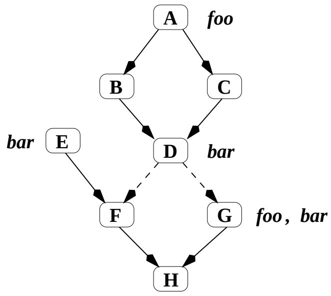


Figure 3: Example Class Hierarchy Graph

We define a binary relation $\approx$ on paths as below.

Definition 3 *Given two paths α and β in the CHG, $\alpha \approx \beta$ iff $fixed(\alpha) = fixed(\beta)$ and $mdc(\alpha) = mdc(\beta)$.*

Obviously, $\approx$ is an equivalence relation on paths. The equivalence relation answers the question of when two different paths identify the same subobject: $\alpha \approx \beta$ iff both α and β identify the same subobject.

Example. Consider the class hierarchy graph in Figure 3. In this example, there are four paths between classes A and H, and the *fixed* part of these paths are as follows: *fixed*(ABDFH) = ABD, *fixed*(ABDGH) = ABD, *fixed*(ACDFH) = ACD, *fixed*(ACDGH) = ACD.

Hence, ABDFH $\approx$ABDGH and ACDFH $\approx$ACDGH. Consequently, both ABDFH and ABDGH denote the same subobject in a H object. Likewise, both ACDFH and ACDGH denote the same subobject in a H object. However, ABDFH $\not\approx$ACDFH. Thus, there are two different subobjects of class A in an instance of H. □

For any path α, let $[\alpha]$ denote the equivalence class to which α belongs. In view of the above explanation, we may use the $\approx$-equivalence classes to identify or name subobjects. Observe that the equivalence relation, $\approx$, can hold only between paths with the same end points, since the condition $fixed(\alpha) = fixed(\beta)$ implies $ldc(\alpha) = ldc(\beta)$. Hence it is meaningful to define:

Definition 4 $mdc([\alpha]) = mdc(\alpha)$; $ldc([\beta]) = ldc(\beta)$

Let us denote the set of all equivalence classes of paths in a CHG G by $\Psi(G)$. We are now ready to define the composition of class instances. The collection of subobjects that constitute an instance of a class X is: $\{ \sigma \in \Psi(G) \mid mdc(\sigma) = X \}$. A subobject σ itself is composed of a collection of members defined by $M[ldc(\sigma)]$.

The Dominance Rule

Definition 5 *We say a path α hides a path β in the CHG if α is a suffix of β. We say a path α dominates a path β in the CHG iff α hides some path $\beta' \approx \beta$.*

Example. In the graph of Figure 3, path GH *hides* ABDGH but not ABDFH. Path GH *dominates* path ABDFH because GH *hides* ABDGH and ABDGH $\approx$ ABDFH. Similarly, FH *dominates* ABDGH since FH *hides* ABDFH and ABDFH $\approx$ ABDGH. □

Lemma 1 *Let $\alpha \approx \alpha'$ and $\beta \approx \beta'$. Then α dominates β iff α' dominates β'.*

Proof. Let $\alpha \approx \alpha'$ and $\beta \approx \beta'$. First note that $\alpha \approx \alpha'$ implies that $fixed(\gamma \bullet \alpha) = fixed(\gamma \bullet \alpha')$, for any γ.

Now assume that α *dominates* β. Then, there exists some path γ such that $\gamma \bullet \alpha \approx \beta$. It is straightforward to show that $\gamma \bullet \alpha' \approx \beta$. Hence α' *dominates* β.

Since α' *dominates* β and $\beta \approx \beta'$, it follows from the definition of domination that α' *dominates* β'. □

Observe that this lemma states that the dominance relation on paths can be meaningfully extended to equivalence classes as below.

Definition 6 *We say that $[\alpha]$ dominates $[\beta]$ if and only if α dominates β.*

Lemma 2 *($\Psi(G)$, dominates) is a partial order.*

Proof.

Reflexivity: Domination is obviously reflexive.

Antisymmetry: Assume that $[\alpha]$ *dominates* $[\beta]$ and $[\beta]$ *dominates* $[\alpha]$. Then, α is a suffix of some $\beta' \approx \beta$. Further, β' is a suffix of some $\alpha' \approx \alpha$. Consequently, $\alpha = \beta' = \alpha'$. Hence, domination is antisymmetric.

Transitivity: Assume that $[\alpha]$ *dominates* $[\beta]$ and $[\beta]$ *dominates* $[\gamma]$. Then, α is a suffix of some $\beta' \approx \beta$. Further, β' is a suffix of some $\gamma' \approx \gamma$. Hence, α is a suffix of some $\gamma' \approx \gamma$. In other words, $[\alpha]$ *dominates* $[\gamma]$. Hence, domination is transitive. □

Formalizing Member Lookup

Let us now turn our attention to member lookup. Let C be a class in a CHG. The set of all subobjects, within an object of class C, that contains a member m is defined as follows.

Definition 7

$$Defns(C, m) = \{\sigma \in \Psi(G) \mid \quad mdc(\sigma) = C \text{ and } m \in Members(ldc(\sigma))\}$$

Example. Consider our example CHG in Figure 3. Which subobjects of a H object have a member `foo`?

$$\mathit{Defns}(H, \texttt{foo}) = \quad \{\{ABDFH, ABDGH\}, \{ACDFH, ACDGH\}, \{GH\}\}$$

Each of these elements of *Defns*(A, `foo`) is an equivalence class of paths. Each of these equivalence classes identifies a unique and distinct subobject of a H object that contains a member called `foo`. Similarly, we have: *Defns*(H, `bar`) = { {EFH}, {DFH, DGH}, {GH} }. □

We are now ready to define the *lookup* operation.

Definition 8 *If A is a set of equivalence classes of paths, we define most-dominant(A) to be the unique element $\sigma \in A$ such that σ dominates σ' for every $\sigma' \in A$, if such a σ exists, and we define most-dominant(A) to be $\perp$ if no such σ exists.*

Definition 9

$$\mathit{lookup}(C, m) = \mathit{most\text{-}dominant}(\mathit{Defns}(C, m))$$

Example. In our example CHG, *lookup*(H, `foo`) = {GH} since {GH} *dominates* every element of *Defns*(H, `foo`). However, *lookup*(H, `bar`) = $\perp$ since *Defns*(H, `bar`) does not have a *most-dominant* element. □

So far, we have used the $\approx$-equivalence classes as a convenient, deterministic representation of subobjects. The algorithm we present later on, however, will manipulate paths and not equivalence classes. Hence, it will be convenient to extend the above definitions to paths as below:

Definition 10

$$\mathit{DefnsPath}(C, m) = \{\alpha \in \mathit{Paths}(G) \mid \mathit{mdc}(\alpha) = C \text{ and } m \in \mathit{Members}(\mathit{ldc}(\alpha))\}$$

Definition 11 *If A is a set of paths, we say $\alpha \in A$ is a most-dominant element of A if α dominates β for every $\beta \in A$.*

Our algorithm, in particular, will return some most-dominant element of $\mathit{DefnsPath}(C, m)$, for a lookup on class C for member mIn effect, rather than return an equivalence class of paths, it will return an arbitrary element of the equivalence class.

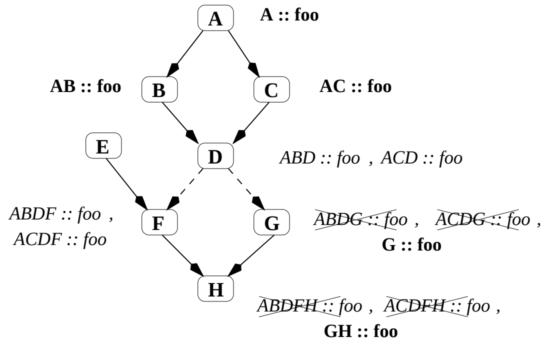


Figure 4: Propagation of definitions of `foo`

4 The Idea Behind the Algorithm

Let us first present the outline of a simple, but inefficient, algorithm that follows directly from the definition of lookup. The algorithm consists of two phases.

The goal of the first phase, the propagation phase, is to compute the sets $\mathit{DefnsPath}(C, \texttt{m})$, for every class C and member `m`. Let us refer to a path α as a *definition*[1] of member `m` if it is an element of $\mathit{DefnsPath}(C, \texttt{m})$ for some C. In the remaining part of this section we will assume that we are only interested in a single member name `m`. (Thus, when we talk about any definition, it may be assumed to be a definition of `m`, unless otherwise mentioned.)

Our goal is to identify all definitions of `m`. Let us refer to a definition α::`m` as an inherited definition if α consists of at least one edge, and as a generated definition otherwise. The set of all generated definitions of `m` is easy to compute: it is simply the set { A:: `m` | $A \in$ N and `m` $\in$ *Members*(A) }. Starting out from the set of all generated definitions, we can identify all inherited definitions through the iterative process of propagating definitions (both generated and inherited) through the CHG: more precisely, a definition α::`m` is propagated along all outgoing edges of $\mathit{mdc}(\alpha)$; the propagation of a definition α::`m` through an edge $X \rightarrow Y$ identifies a new (inherited) definition, namely $\alpha \bullet (X \rightarrow Y)$::`m`. This iterative process stops when there are no more new definitions to propagate.

This propagation phase lets us identify the set of definitions that "reach" each node. The second phase of the algorithm is to simply determine for every class C if the set of definitions of `m` that reach C has a *most-dominant* definition α. If it does, then the result of *lookup*(C,`m`) is α. If not, then *lookup*(C,`m`) is undefined.

[1] The term "definition" and several other subsequent terms are used to draw an analogy between the algorithm presented here and the standard "reaching definitions" problem [3, Page 610]. However, as will become clear, there are quite a few differences between the two problems, and the reader should not be misled by the analogy.

Example. Consider Figure 4, which illustrates this process for member `foo`. (Figure 5 illustrates the same for member `bar`.) There are two generated definitions of `foo` namely, A::`foo` and G::`foo`. Propagation of A::`foo` creates the two definitions AB::`foo` and AC::`foo`. Propagation of these two definitions, in turn, creates the two definitions ABD::`foo` and ACD::`foo`, both of which reach the same node D. The set of definitions that reach a node are shown adjacent to the node. If the set of reaching definitions at a node has a most-dominant definition, that definition is shown in bold font. The other definitions are shown in italic. Obviously, the lookup is unambiguous exactly for the nodes with a reaching definition in bold font. Other aspects of the figures (relating to nodes G and H) will be explained later. □

Optimizing the Propagation Phase

Note that our ultimate goal is to just identify the most-dominant reaching definition at every node. Not surprisingly, it turns out that we do not really have to propagate all the definitions in order to do this.

Consider the example in Figure 4. The definitions ABDG::`foo` and ACDG::`foo` reach the node G, which generates its own definition G::`foo`. It turns out that it is unnecessary to propagate ABDG::`foo` and ACDG::`foo` out of node G because they are "killed" by G::`foo`. The following lemma indicates when it is valid to "kill" a definition, i.e. , not propagate it:

Lemma 3 *Path* $\gamma\bullet(X \to Y)$ *dominates path* $\delta\bullet(X \to Y)$ *iff path* γ *dominates path* δ. [2]

Proof. ($\Rightarrow$) Assume that $\gamma \bullet (X \to Y)$ dominates path $\delta \bullet (X \to Y)$. Then, $\gamma \bullet (X \to Y)$ is a suffix of some path $\alpha \approx \delta \bullet (X \to Y)$. Clearly, α must be of the form $\beta \bullet \gamma \bullet (X \to Y)$. Further, we must have $\beta \bullet \gamma \approx \delta$. Hence, γ dominates δ.

($\Leftarrow$) Assume that γ dominates δ. Then, γ is a suffix of some path $\alpha \approx \delta$. Clearly, $\gamma \bullet (X \to Y)$ is a suffix of $\alpha \bullet (X \to Y)$, and $\alpha \bullet (X \to Y) \approx \delta \bullet (X \to Y)$. Hence, $\gamma \bullet (X \to Y)$ dominates $\delta \bullet (X \to Y)$. □

[2] Lemma 3 can also be stated as:

$$\text{most-dominant}(\{\, \gamma \bullet (X \to Y), \delta \bullet (X \to Y) \,\}) = \text{most-dominant}(\{\, \gamma, \delta \,\}) \bullet (X \to Y)$$

(where $\perp \bullet (X \to Y)$ is defined to be $\perp$). The member-lookup problem is a pseudo-meet-over-all-paths dataflow analysis problem, where the pseudo-meet operation is "most-dominant" and the transfer function associated with edges is given by the path extension operator $\bullet$. The above equation says that the path extension operator is distributive over "most-dominant". This implies that the problem is a distributive dataflow analysis problem. Hence, it is not necessary to propagate *all* the definitions reaching a node X along the outgoing edges of X; it is sufficient to propagate just the *meet* of all the reaching definitions.

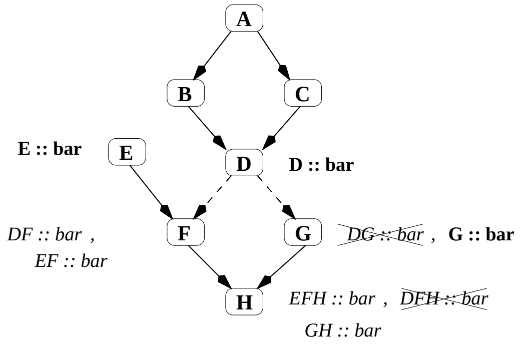


Figure 5: Propagation of definitions of `bar`

Corollary 1 *If* γ *dominates* δ *and* $\gamma \neq \delta$, *then for any path* $\gamma \bullet \omega$ *and any set* S *containing path* $\gamma \bullet \omega$, S *has a most-dominant element iff* $S - \{\, \delta \bullet \omega \,\}$ *has a most-dominant element, in which case the most-dominant elements of both sets are* $\approx$*-equivalent.*

It follows from the above corollary that if γ and δ are two definitions of some member `m` that reach a node X and that if γ dominates δ then we may kill δ at this node. The reason is as follows. Let ω be any path from node X to some node Y. Killing δ at node X may prevent definition $\delta \bullet \omega$ from reaching node Y. Hence, the set of reaching definitions of `m` at node Y will be affected. *However, the above corollary implies that the most-dominant reaching definition of* `m` *at node Y will still be the same (upto* $\approx$*-equivalence).* In other words, the result of the lookup will not be affected.

Example. If a class X has its own definition of `m`, then clearly the generated definition X::`m` trivially dominates any other definition α::`m` that reaches X. Hence, we may kill α::`m` at node X. Thus, G::`foo` kills $ABDG$::`foo` and $ACDG$::`foo` in Figure 4. This is similar to what happens in the standard reaching definitions problem.

However, consider node H. No definition of foo is generated at H, but we have three reaching definitions, namely GH::foo, which reaches H along one edge $G \to H$, and $ABDFH$::foo and $ACDFH$::foo, which reach H along another edge $F \to H$. Since GH dominates $ABDFH$ and $ACDFH$, definitions $ABDFH$::foo and $ACDFH$::foo can be killed at node H. This kind of killing does not happen in the reaching-definitions problem, but is valid in the member-lookup algorithm. Killed definitions are shown crossed out in Figures 4 and 5. □

Let us now explain how exactly our algorithm incorporates killing. In the naive algorithm, as described earlier, the second phase (identifying the most-dominant element among all reaching definitions for every node) was performed after the first phase (propagation of all definitions). For purposes of killing, it is convenient to interleave the two phases. In particular, once the set of reaching definitions of a node X has been determined, we scan this set to check if it has a most-dominant element. Any definition found to be dominated by some other definition during this scan will be killed immediately. All other definitions will be propagated out of the node X.

In particular, for a node X for which the lookup is well-defined (i.e., unambiguous), the set of reaching definitions has a most-dominant element and we need to propagate only this definition along the outgoing edges of X. Let us call the definitions so propagated **red** definitions. For a class X for which the lookup is ambiguous, one or more definitions may be propagated along the outgoing edges of X. Let us call the definitions so propagated **blue** definitions. More formally:

Definition 12 *A definition α of* m *is said to be a red definition of* m *iff for every prefix γ of α such that $\gamma \neq \alpha$, γ is a most-dominant element of $DefnsPath(mdc(\gamma), \mathtt{m})$.*

A question may possibly arise in the mind of the reader at this point. Why do we need to propagate the blue definitions at all? Could we not use Lemma 3 to justify killing all the blue definitions as below? Let α be any definition that reaches a node X for which the lookup is ambiguous. Then, there exists some other definition β that reaches X such that α does not dominate β. Assume we propagate α along some path ω from X to some other class Y, giving us the definition $\alpha \bullet \omega$. Note that $\alpha \bullet \omega$ *cannot* be the most-dominant reaching definition of m at Y because it does not dominate $\beta \bullet \omega$ (from Lemma 3). In other words, the result of the lookup at Y cannot be $\alpha \bullet \omega$. So, why can't we simply kill definition α at node X?

Unfortunately, the above argument is not complete. The above argument does point out correctly that a blue reaching definition cannot be the most-dominant reaching definition. However, a blue reaching definition *may determine* if some red definition is the most-dominant reaching definition or not. The example in Figure 5 illustrates this. *lookup*(F,`bar`) is ambiguous, with two reaching definitions EF and DF. Here, *lookup*(H,`bar`) is also ambiguous because GH::`bar` does not dominate EFH::`bar`. If blue EF is not propagated from F to H, however, it might appear as though *lookup*(H,`bar`) was unambiguous. (In the case of member `foo`, as shown in Figure 4, the lookup at node F is ambiguous, but the lookup at the subsequent node H is not ambiguous.) This is the reason for propagating blue definitions.

One advantage of killing definitions is immediately obvious: the propagation phase itself has to do less work, and the set of reaching definitions at many nodes may end up being smaller, speeding up the second phase too. Killing definitions has another significant advantage which will become clear later: it allows for an efficient domination check among reaching definitions.

Identifying the Most Dominant Definition

It is fairly straightforward to identify the most-dominant definition, if one exists, of the set of reaching definitions. It is similar to selecting the maximum element from a list of integers, with minor modifications to handle the fact that dominance is only a partial order and not a total order. The only non-trivial operation is that of checking for domination between two paths. We explain how this can be done below.

As we saw earlier, the reaching definitions are of two kinds, the red definitions and the blue definitions. We observed earlier that a blue definition is guaranteed *not* to be the most-dominant element. This implies that we may select the most-dominant definition among the red reaching definitions, if one exists, and simply verify that it dominates all the blue reaching definitions.

In particular, this implies that we will not have to test for domination between two blue reaching definitions. Thus, we will need to check for dominance among two reaching definitions α and β *only if α and β reach the node under consideration along different edges* (since if multiple definitions are propagated along a single edge, they will all be blue definitions). This is interesting because it allows us to implement the test for domination efficiently, as explained below.

Definition 13 *A path α is said to be a virtual path (v-path) if it contains at least one virtual edge.*

Let Ω be some new symbol not in the set of classes N. Let N_Ω denote the union $N \cup \{\, \Omega \,\}$. We define a function *leastVirtual* that maps paths in the CHG to N_Ω as follows.

Definition 14

$$\begin{aligned} leastVirtual(\beta) &= mdc(fixed(\beta)) && \text{if } \beta \text{ is a v-path} \\ &= \Omega && \text{if } \beta \text{ is not a v-path} \end{aligned}$$

Lemma 4 *Let X and Y be two different direct base classes of Z. Further, let $\alpha \bullet (X \to Z)$, $\beta \bullet (Y \to Z) \in DefnsPath(Z, \mathtt{m})$. If $\alpha \bullet (X \to Z)$ is a red definition, then $\alpha \bullet (X \to Z)$ dominates $\beta \bullet (Y \to Z)$ iff either*
(i) $leastVirtual(\beta)$ is a virtual base of $ldc(\alpha)$ or
(ii) $leastVirtual(\beta) = leastVirtual(\alpha) \neq \Omega$.

Proof.
Let α' denote $\alpha \bullet (X \to Z)$ and let β' denote $\beta \bullet (Y \to Z)$.

($\Rightarrow$) We are given that α' dominates β'. Hence, α' is a suffix of some $\gamma \bullet \alpha' \approx \beta'$. Let γ' denote $\mathit{fixed}(\beta')$ $= \mathit{fixed}(\gamma \bullet \alpha')$. There are two cases to consider.

Consider the case that γ' is a proper prefix of γ. Then, clearly, $\mathit{leastVirtual}(\beta')$ ($= mdc(\gamma')$) must be a virtual base of $ldc(\alpha')$.

Consider the case that γ is a prefix of γ'. Clearly, in this case $\mathit{leastVirtual}(\beta') = mdc(\gamma') = \mathit{leastVirtual}(\alpha') \neq \Omega$.

($\Leftarrow$) Assume that $\mathit{leastVirtual}(\beta)$ is a virtual base of $ldc(\alpha)$. In other words, there exists a path γ from $\mathit{leastVirtual}(\beta)$ to $ldc(\alpha)$ whose first edge is a virtual edge. Then, let δ denote the path $\mathit{fixed}(\beta)\ \gamma\ \alpha'$. Clearly, α' hides δ and $\delta \approx \beta'$. Hence, α' dominates β'.

Now assume that $\mathit{leastVirtual}(\beta) = \mathit{leastVirtual}(\alpha) \neq \Omega$. Clearly, $\mathit{fixed}(\beta)$ and $\mathit{fixed}(\alpha)$ are both elements of $\mathit{DefnsPath}(\mathit{leastVirtual}(\beta), \mathtt{m})$. The definition of a red definition implies that $\mathit{fixed}(\alpha)$ dominates (in fact, hides) $\mathit{fixed}(\beta)$. It follows that α' dominates β'. □

Abstracting Paths

We are now ready to present our algorithm in detail. It follows from Lemma 4 that we do not really need full information about paths to determine if one path dominates another path. Hence, our algorithm propagates not paths but *abstractions of paths*.

Let us first consider blue definitions. As explained earlier, the only operation we perform on a blue definition is to check if it is dominated by some other red definition. It follows from Lemma 4 that for any blue definition β it would be sufficient to propagate $\mathit{leastVirtual}(\beta)$.

Let us be more precise about what it means to propagate $\mathit{leastVirtual}(\beta)$. When a definition β is propagated through an edge $B \rightarrow D$ it creates the definition $\beta \bullet (B \rightarrow D)$. Hence, when we propagate the abstraction $\mathit{leastVirtual}(\beta)$ through the edge $B \rightarrow D$ we need to create the abstraction $\mathit{leastVirtual}(\beta \bullet (B \rightarrow D))$. We now define an operation $\circ$ that abstracts the path concatenation operator $\bullet$, which allows us to do this.

Definition 15

$$X \circ (B \rightarrow D) \;=\; \begin{array}{l} \text{if } (X \neq \Omega) \text{ then } X \\ \text{else if } B \rightarrow D \in E_v \text{ then } B \\ \text{else } \Omega \end{array}$$

Note that,
$\mathit{leastVirtual}(\beta \bullet (B \rightarrow D)) = \mathit{leastVirtual}(\beta) \circ (B \rightarrow D)$.

The above abstraction of blue definitions is a critical step in improving the efficiency of the algorithm. With this abstraction, we need to propagate only subsets of N_Ω (whose size can be at most $|N| + 1$) instead of propagating subsets of $\mathit{Paths}(G)$ (whose sizes can be exponential in $|N|$).

Now let us consider red definitions. With a red definition we may need to check both if it dominates some other definition as well as if it is dominated by some other definition. In view of Lemma 4, we may abstract a red definition α to the pair $(ldc(\alpha), \mathit{leastVirtual}(\alpha))$. Further, when such a pair (L, V) is propagated through an edge $B \rightarrow D$, we simply transform it into $(L, V \circ (B \rightarrow D))$. A red definition $r_1 = (L_1, V_1)$ dominates another red definition $r_2 = (L_2, V_2)$ iff r_1 and r_2 satisfy the condition in Lemma 4, *i.e.*, iff either V_2 is a virtual base of L_1 or $V_1 = V_2 \neq \Omega$.

Though our algorithm does not require full path information, it may be convenient to return full path information for a successful member lookup, since compilers may need the full path information to generate code for the lookup. If this is desired, we may "abstract" a red definition α to the triple $(ldc(\alpha), \mathit{leastVirtual}(\alpha), \alpha)$. (The first two components are still desirable to enable a quick dominance test.) This can be done without affecting the complexity of the algorithm (since at most one red definition is propagated across an edge).

Example. For our example, Figures 6 and 7 illustrate the propagation of path abstractions for definitions of `foo` and `bar` respectively in the CHG. At each node, the left hand side of $\Rightarrow$ represents the path abstractions that reach the node and the right hand side of $\Rightarrow$ corresponds to the abstraction produced at the node by our algorithm, as explained above. For example, consider node D in Figure 6. The node has two reaching red definitions, both of which have the identical abstraction (A,Ω). Since (A,Ω) does not dominate (A,Ω), the lookup is ambiguous at D. Hence, the red definitions become blue definitions for purposes of further propagation. Consequently, they are abstracted into the singleton Ω, which is further transformed into D by propagation along $D \rightarrow F$ (using the $\circ$ operation). □

5 The Algorithm and Its Complexity

A complete description of our algorithm appears in Figure 8. It is based on a traversal of the CHG in topological sort order: if `B` is a base class of `D` then `B` will be visited and processed before `D`. For every class C and every relevant member `m`, our algorithm computes the value of *lookup*[C,`m`], which is either **Red** D, where $D \in N \times N_\Omega$ or **Blue** S, where $S \subseteq N_\Omega$.

The value **Blue** S implies that the corresponding lookup was ambiguous (with S being an abstraction of the set of definitions that created the ambiguity),

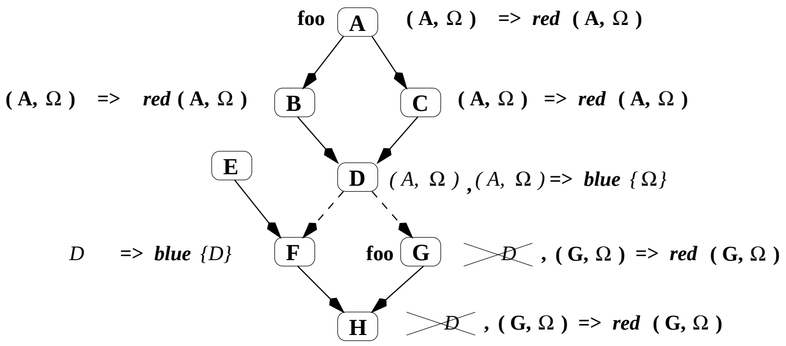


Figure 6: Propagation of definitions of `foo`

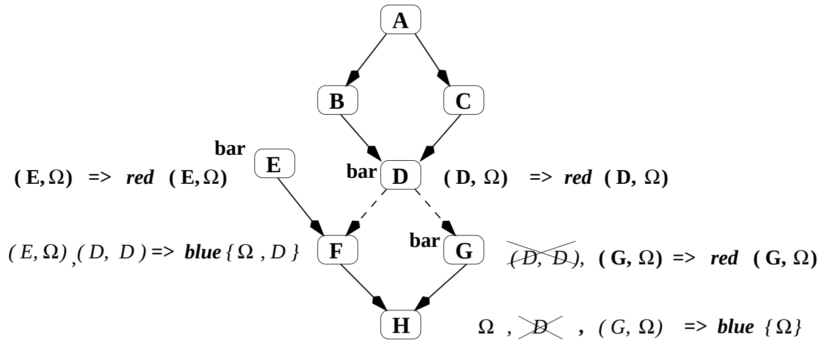


Figure 7: Propagation of definitions of `bar`

while the value **Red** D implies that the corresponding lookup was unambiguous (with D being the abstraction $(ldc(\alpha), leastVirtual(\alpha))$ of the definition α to which the lookup resolved).

Note that the algorithm, as described, eagerly "tabulates" the lookup operation: it constructs a table `lookup` indexed by a class and a member name; once the table has been constructed, every lookup operation takes constant time. It is easy enough to modify the algorithm into a "memoizing lazy" algorithm that does not compute table entries that are unnecessary: a request for *lookup*[C,`m`] will recursively invoke *lookup*[B,`m`] for every direct base class B of C if necessary; as long as the algorithm caches or memoizes the results of every lookup performed, this will not worsen the complexity of the algorithm.

Let us now understand the complexity of the algorithm. Assume that the test for whether a class is a virtual-base of another class can be implemented in constant time. (Note that to implement the test for whether a class is a virtual-base of another class in constant time, we can construct a boolean matrix using a "transitive closure"-like algorithm. A straight-forward implementation of this will take time $O(|N| \times (|N| + |E|))$. Note that a compiler requires this information, in some form, and will have to compute it any way.)

If the above "preprocessing" has been done, what is the worst-case complexity of a single lookup operation?

Let us first consider the simpler, and hopefully common, case where the result of every possible lookup operation is unambiguous. In this case, lines [30]-[32] are never executed. At most one definition reaches a node through each incoming edge, and the lookup at a particular class can be done in time proportional to the number of incoming edges it has. The whole lookup is, hence, linear in the size of the class hierarchy graph, i.e. $O(|N| + |E|)$, where $|N|$ is the number of nodes in the CHG and $|E|$ is the number of edges in the CHG. (This assumes that the membership test of line [12] can be done in constant time. If not, the cost of $O(|N|)$ membership tests will have to be added to the complexity.)

In the general case, however, the worst-case complexity of a single lookup can be $O((|N| + |E|) \times |N|)$. This is because $\Theta(|N|)$ blue definitions can reach a class though each incoming edge. Hence, the cost of performing the unions in lines [30]-[32] can be $\Theta(|N|)$ for every edge in the graph.

The worst-case complexity of doing all possible lookups for a single member name is the same as that of doing a single lookup. If $|M|$ denotes the number of member names in the program, then the worst-case complexity

```
function dominates ( (L1,V1), (L2,V2) ) {
    return (V2 ∈ virtual-bases[L1]) or (V1 = V2 ≠ Ω);
}

procedure doLookup() {
    for each class C in topological sort order {
        // Identify list of members for C
        Members[C] := M[C];
        for every direct base class X of C
            Members[C] := Members[C] ∪ Members[X];
        // Identify dominating definition for each member of C
        for every m ∈ Members[C] {
            if (m ∈ M[C]) then
                lookup[C,m] := Red (C,Ω)
            else {
                toBeDominated := ∅;        noCandidate := true;
                for every direct base class X of C {
                    if (m ∈ Members[X]) then {
                        case lookup[X,m] of
                            Red (L,B) ⇒
                                V := B ∘ (X → C);
                                if noCandidate then {
                                    noCandidate := false;
                                    (candidateL, canditateV) := (L,V);
                                } else if dominates((L,V), (candidateL, canditateV)) then {
                                    (candidateL, canditateV) := (L,V);
                                } else if ( not dominates ((candidateL, canditateV), (L,V)) then
                                    toBeDominated := toBeDominated ∪ {canditateV, V} ;
                                    noCandidate := true;
                                }
                            Blue (S) ⇒
                                for every B ∈ S
                                    toBeDominated := toBeDominated ∪ { B ∘ (X → C) };
                    }
                }
                if (noCandidate) then
                    lookup[C,m] := Blue (toBeDominated)
                else {
                    for every B ∈ toBeDominated {
                        if ( (B ∈ virtual-bases[candidateL]) or (B = candidateV ≠ Ω)) then
                            remove B from toBeDominated
                    }
                    if (toBeDominated = ∅)
                        then lookup[C,m] := Red (candidateL, candidateV)
                        else lookup[C,m] := Blue (toBeDominated ∪ {candidateV})
                }
            }
        }
    }
}
```

Figure 8: The Member Lookup Algorithm

of constructing the whole lookup table, as described in Figure 8 is $O(|M| \times (|N| + |E|) \times |N|)$ in the general case, and $O((|M| + |N|) \times (|N| + |E|))$ in the (possibly common) case of a program in which every table entry is unambiguous.

6 Other Issues

The C++ rules for member lookup, in reality, are slightly more complicated than what we have presented so far. In particular, members of a class may be classified into static[3] and non-static members. While our earlier presentation is valid in the absence of static members, it must be modified as below to handle static members.

Definition 16 *If A is a set of equivalence classes of paths, we define maximal(A) as follows:*

$$maximal(A) = \{\sigma \in A \mid \nexists \sigma' \in A. \quad (\sigma' \neq \sigma \text{ and } \sigma' \text{ dominates } \sigma)\}$$

Definition 17 *We define lookup(C, m) to be maximal(Defns(C, m)), if*

1. $|maximal(Defns(C, m))| = 1$, *or*
2. $\forall \sigma_1, \sigma_2 \in maximal(Defns(C, m))$
   (a) $ldc(\sigma_1) = ldc(\sigma_2)$, *and*
   (b) *m is a static member of* $ldc(\sigma_1)$.

lookup(C, m) is defined to be $\perp$ otherwise.

Observe that if condition (ii) holds true in the above definition, then the lookup can return a set with more than one element. In terms of an implementation, however, it is sufficient if some representative element of this set is returned.

It is fairly straightforward to extend our algorithm to deal with static members. We modify the function *dominates* defined in lines [1]-[3] of Figure 8 to take the member name m as an extra argument. The function returns true iff (1) $V2 \in$ *virtual-bases*[$L1$] **or** (2) V1 = V2 $\neq \Omega$ **or** (3) (L1 = L2) and m is a static member of L1.

There are a couple of other features of C++ related to member lookup that we briefly discuss below.

One of these features is that of *access rights.* The access rights rules of C++ specify which members of a base class may be accessed in the scope of the derived class. The access rights do not *affect* the member lookup process in any way; they are applied only *after* a "successful" member lookup to determine if that particular member access is legal. We show in [8] how our lookup algorithm can be extended in a straightforward way to compute access rights.

Another related issue is the resolution of *unqualified* names. The problem discussed in this paper concerns the resolution of *qualified* names, such as the name `m` in the expressions `x.m` or `X::m`. However, names may also occur unqualified in a C++ program. For example, the names `a` and `b` in the expression `a+b`, are unqualified names. The resolution of an unqualified name in C++ is essentially the same as the traditional name lookup process in the presence of nested scopes. The only complication is that any of these nested scopes may itself be a *class*, and the "local" lookup within a class scope itself reduces to the member lookup problem addressed in this paper. More details about this appear in [8].

7 Related Work

7.1 Member Lookup in C++

The work most closely related to our work is that of Rossie and Friedman [9], who present a formalism that models the multiple inheritance mechanism of C++. They show how, given a class hierarchy γ, one may define a subobject poset $(\Sigma(\gamma), \leq_{so})$. Let us denote the class hierarchy graph of a class hierarchy γ by CHG(γ). We can show that:

Theorem 1 *The poset $(\Psi(CHG(\gamma))$, dominates) is isomorphic to the poset $(\Sigma(\gamma), \leq_{so})$.*

Thus, our $\approx$-equivalence classes directly correspond to the subobjects defined by Rossie and Friedman.

Rossie and Friedman utilize their subobject poset to define two member lookup operations, *dyn* and *stat*, that essentially model the lookups performed for virtual and non-virtual members respectively. However, these lookup operations, for a given member `m`, are defined as partial functions that map *subobjects* to *subobjects.* Recall that the lookup operation defined in this paper, for a given member `m`, maps a *class* to a *subobject.* The Rossie-Friedman lookup can be defined in terms of our lookup operation as below[4]:

$$\begin{aligned} \text{dyn}(m, \sigma) &= lookup(mdc(\sigma), m) \\ \text{stat}(m, \sigma) &= (lookup(ldc(\sigma), m)) \bullet \sigma \end{aligned}$$

[3]It is also possible to introduce new type names and enumeration constants into the scope of a class. For purposes of member lookup, these are treated exactly like static members.

[4]The Rossie-Friedman operations correspond to a C++ implementation where member lookup is done completely at run-time. These equations show how the lookup can be staged such that most of the work is done at compile time, with the run-time operation being a constant-time operation (as is done in typical C++ implementations). Our lookup operation models the part of member lookup that is typically performed statically by a compiler.

where the "subobject composition" operator $\bullet$ is defined by: $[\alpha] \bullet [\beta] = [\alpha \bullet \beta]$

Tip *et al.* [12] address a different problem, that of class hierarchy slicing, but their work is *based* on the Rossie and Friedman formalism. They present a slightly altered version of the Rossie and Friedman member lookup definition, but one that is still based on the subobject graph.

The definitions presented by both Rossie and Friedman and Tip *et al.* are executable definitions, providing us with member lookup algorithms. However, direct implementations of these definitions can be inefficient, as we will discuss shortly.

As an example of how current C++ compilers implement member lookup, we studied[5] the implementation in GNU's g++ compiler (version 2.7.2.1), whose source code is publicly available.

The lookup algorithm in g++ is based on a breadth-first traversal of the subobject graph. In particular, the lookup for a member `m` in a class X begins at the node in the subobject graph corresponding to an object of class X. If class X itself does not have a member called `m`, the algorithm performs a scan of all the "subobjects" of an X object, in breadth-first order, and attempts to identify the most-dominant definition of `m`. Thus, the g++ implementation is more or less a direct implementation of the Rossie and Friedman definition (though it predates the Rossie and Friedman formalism), with the exception that the g++ implementation of selecting the most-dominant definition is incorrect. The g++ algorithm keeps track of the "most-dominant" member found by the traversal. When the traversal visits a subobject, it checks to see if that subobject has a member called `m`. If it does, it checks to see if either the most-dominant member found so far or the newly found member dominates the other. If one of the two dominates the other, it keeps that definition around as the most-dominant one. *If neither definition dominates the other one, the algorithm reports ambiguity and quits.*

It is the step described last, where ambiguity is reported if neither definition dominates the other one, that is incorrect. It is possible, when using a breadth-first search, to encounter definitions d_1 and d_2, neither of which dominates the other, and to then later encounter a definition d_3 which dominates both d_1 and d_2. This happens in the example presented in Figure 7.1. Though the lookup in line [s2] is unambiguous, the g++ compiler flags it as being ambiguous[6].

Let us briefly discuss the complexity of the algorithms described above. In the worst case, the subobject graph's size can be exponential in the size of the class hierarchy graph and, hence, all the algorithms mentioned above have a worst-case complexity that is exponential in the size of the class hierarchy graph, while the complexity of our algorithm ranges from linear to quadratic in the size of the class hierarchy graph. (For the kind of class hierarchies that arise in practice, however, this kind of exponential blowup in the size of the subobject graph does not seem to occur. Hence, in practice, we do not expect our algorithm to *exponentially* outperform the algorithms described above. But we do expect that our algorithm will perform as well or better than these algorithms. Since the time spent on member lookups in a compiler can be as much as 15% of the total compilation time [11], faster lookups could be of relevance to a compiler.)

```
struct S { int m; };
struct A: virtual S { int m; };
struct B: virtual S { int m; };

struct C: virtual A, virtual B { int m; };
struct D: C {};
struct E: virtual A, virtual B, D {};

main() {
    s1: E e;
    s2: e.m = 10;
}
```

Figure 9: A counterexample for the g++ algorithm.

An informal discussion of subobject graphs and multiple inheritance in C++ appears in [5] and [10], but neither provides a formal model for multiple inheritance or an algorithm for member lookup.

7.2 Member Lookup in Other Languages

Though there exist other languages that support multiple inheritance, we are not aware of other languages that have concepts such as dominance or two different kinds of inheritance (namely, virtual and non-virtual). Hence, member lookup in C++ appears to be quite distinct from member lookup in other languages.

Self [2] is an example of another language that supports multiple inheritance. Member lookup and ambiguity checking in Self, however, are conceptually much simpler than in C++. Self does not have the concept of *classes*. Instead, objects directly inherit from other objects. A member name `m` is unambiguous in a given object iff exactly one definition of `m` is *visible* in that object. (A member `m` in a base object is said to be visible

[5]Thanks to Mike Stump, co-author of g++, for confirming our understanding of the g++ implementation.

[6]In fact, 3 of the 7 compilers we tried this example on reported this lookup as being ambiguous.

in a derived object iff there exists an inheritance path between the two objects that does not contain any other object with a member called `m`.) Member lookup in Self is done completely at run-time, and, hence, the speed of the lookup greatly affects the performance of Self programs. Hence, several techniques have been developed to optimize the lookup at run-time [7, 6]. These techniques, however, are not directly relevant to compile-time member lookup.

Attali et al. [4] present a semantics and algorithm for lookup in Eiffel, another language with multiple inheritance. Member lookup in Eiffel is complicated by the presence of a feature called *renaming*, that allows a derived class to rename an inherited member.

The Attali et al. algorithm, however, assumes that the input program is "statically well typed". In particular, they assume that none of the lookups in the source program is ambiguous. It is worth pointing out here that much of the complexity of member lookup in C++ is in identifying ambiguous lookups. If one assumes that a particular lookup is unambiguous, then the lookup can be done very simply as follows. Associate each class X with a topological number, $\textit{top-sort}(X)$, such that the topological number of a base class is less than the topological number of a derived class. (Since a compiler sees a base class' definition before it sees a derived class' definition, this numbering can be done trivially.) Then, from the set of definitions that reach a class X, one simply selects the σ for which $\textit{top-sort}(\textit{ldc}(\sigma))$ is maximum as the most dominant definition.

8 Conclusions

We believe that C++ has a fairly complicated semantics, thanks to the presence of a large number of features, not necessarily all orthogonal. This complexity warrants taking a reasonably formal approach towards building C++ implementations.

The main contribution of this paper is an algorithm for member lookup in C++. A secondary contribution of this paper is a reformulation of the Rossie-Friedman multiple inheritance formalism, which may be of use as a formal basis for deriving efficient implementations of a C++ compiler. The formalism, however, addresses only certain aspects of C++. A similar formal treatment of other aspects of C++ implementation would be worth pursuing.

Acknowledgements We thank John Field, Michael Burke, Ron Cytron, Frank Tip, and the anonymous referees for their comments which greatly helped improve the paper.

References

[1] I. P. S. Accredited Standards Committee X3. Working paper for draft proposed international standard for information systems—programming language C++. Draft of 26 September 1995.

[2] O. Agesen, L. Bak, C. Chambers, B. W. Chang, U. Hölzle, J. Maloney, R. B. Smith, D. Ungar, and M. Wolczko. The Self 4.0 Programmer's Reference Manual. Sun Microsystems, Inc.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[4] I. Attali, D. Caromel, and S. O. Ehmety. A Natural Semantics for Eiffel Dynamic Binding. *ACM Trans. on Programming Languages and Systems*, 18(6):711–720, 1996.

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

[6] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. of the European Conf. on Object-Oriented Programming*, July 1991.

[7] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proc. of the SIGPLAN'94 Conf. on Programming Language Design and Implementation*, pages 326–336, June 1994.

[8] G. Ramalingam and H. Srinivasan. A Member Lookup Algorithm for C++. Technical report, IBM, 1997. In Preparation.

[9] J. G. Rossie and D. P. Friedman. An Algebraic Semantics of Subobjects. In *Proc. of Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–199, October 1995.

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

[11] B. Stroustrup, 1996. Personal Communication.

[12] F. Tip, J. D. Choi, J. Field, and G. Ramalingam. Slicing Class Hierarchies in C++. In *Proc. of Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 179–197, October 1996.