Universitetet i Oslo
Institutt for informatikk

Incremental Reasoning for Multiple Inheritance

Johan Dovland,
Einar B. Johnsen,
Olaf Owe, and Martin Steffen

Research Report 373
ISBN 82-7368-333-8

April 2008

Incremental Reasoning for Multiple Inheritance

Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen

Department of Informatics, University of Oslo
PO Box 1080 Blindern, NO-0316 Oslo, Norway
{johand,einarj,olaf,msteffen}@ifi.uio.no

Abstract. Object-orientation supports code reuse and incremental programming. Multiple inheritance increases the power of code reuse, but complicates the binding of method calls and thereby program analysis. Behavioral subtyping allows program analysis under an *open world assumption*; i.e., under the assumption that class hierarchies are extensible. However, method redefinition is severely restricted by behavioral subtyping, and multiple inheritance often leads to conflicting restrictions from independently designed superclasses. This paper presents an approach to incremental reasoning for multiple inheritance under an open world assumption. The approach, based on a notion of *lazy behavioral subtyping*, is less restrictive than behavioral subtyping and fits well with multiple inheritance, as it incrementally imposes context-dependent behavioral constraints on new subclasses. We formalize the approach as a calculus, for which we show soundness.

1 Introduction

Object-orientation supports code reuse and incremental programming through inheritance. Class hierarchies are extended over time as subclasses are developed and added. A class may reuse code from its superclasses but it may also specialize and adapt this code by contributing new method definitions, possibly overriding definitions in superclasses. This way, the class hierarchy allows programs to be represented in a compact and succinct way, significantly reducing the need for code duplication. *Late binding* is the underlying mechanism for this incremental programming style; the binding of a method call at runtime depends on the actual class of the called object. Consequently, the code to be executed depends on information which is not statically available. Although late binding is an important feature of object-oriented programming, this loss of control severely complicates reasoning about object-oriented programs.

Behavioral subtyping is the most prominent solution to regain static control of late bound method calls (e.g., [1, 16, 17]). This approach achieves incremental reasoning with an *open world assumption*; i.e., class hierarchies are extensible. However, the approach restricts how methods may be redefined in subclasses. To avoid reverification, any method redefinition must *preserve* certain properties of the method being redefined. In particular, this applies to the method's contract; i.e., the pre- and postcondition for its body. The contract can be seen as a description of the promised behavior of all implementations of the method. Unfortunately, this restriction hinders code reuse and is often violated in practice [23]; e.g., it is not respected by the standard Java library definitions.

Work on behavioral reasoning about object-oriented programs has mostly focused on languages with single inheritance (e.g., [5, 20, 21]). With single inheritance, a class

is derived from one direct superclass. Recently, we have relaxed the behavioral subtyping restrictions on method redefinition in this context, leading to a notion of *lazy behavioral subtyping* [10]. Given a method m declared with precondition p and postcondition q, there is no need to restrict the behavior of methods overriding m to ensure that these adhere to this specification. Instead, it suffices to preserve the "part" of p and q *used to verify* the program at the current stage. Specifically, if m is used in a method call $\{r\}\ e.m(\ldots)\ \{s\}$, the pre- and postconditions r and s at that call-site constitute m's *required* behavior. Only these weaker conditions need to be preserved to avoid reverification. Behavioral subtyping is not implied by this approach: When m is overridden in a class C, the new definition need not implement all superclass specifications of m, but only the requirements made by superclasses of C towards usage of m. Lazy behavioral subtyping still supports incremental reasoning under an open world assumption.

Multiple inheritance allows a class to be derived from several direct superclasses. This offers greater flexibility than single inheritance, as several class hierarchies can be combined in a subclass. It also complicates language design and is often explained in terms of complex run-time data structures such as virtual pointer tables [24]. Formal treatments are scarce (e.g., [4, 6, 11, 22, 25]) but help clarify intricacies, thus facilitating design and reasoning for programs using multiple inheritance. Multiple inheritance also complicates behavioral reasoning, as name conflicts may occur between methods independently defined in different branches of the class hierarchy. Although name conflicts are often resolved through qualification or renaming [2, 18, 24], it might be undesirable to force the programmer to modify method names, making programs more difficult to understand and maintain. Name conflicts may also be seen as a natural feature of multiple inheritance and resolved by imposing an ordering of superclasses [7, 9, 14].

In this paper, we extend the lazy behavioral subtyping approach to multiple inheritance class-based systems. For this purpose, we adopt a *pruned binding* strategy for method calls, which resolves name conflicts based on the static context of each method call. This binding strategy supports incremental reasoning and ensures that method binding applies inside the scope of the constraints ensured by lazy behavioral subtyping. The approach is formalized as an inference system, for which we show soundness.

Paper overview. Section 2 introduces late binding and multiple inheritance, and Section 3 proof environments for behavioral reasoning. Section 4 presents the inference system for incremental reasoning. Section 5 discusses related work, and Section 6 concludes the paper.

2 Late Binding and Multiple Inheritance

In an object-oriented program, classes are related by means of inheritance in a class hierarchy. We say that a class C_1 is *below* another class C_2 (and write $C_1 \leq C_2$) if C_1 extends C_2, C_1 extends a class which is below C_2, or C_1 and C_2 are the same class. Furthermore, C_2 is *above* C_1 if C_1 is below C_2. A *subclass* is below a *superclass*. Single inheritance class hierarchies are tree-shaped; a class can have several direct subclasses, but only one direct superclass. *Vertical* name conflicts occur when a method is overridden in a subclass. The *binding strategy* for method calls must resolve such conflicts.

$$
\begin{array}{ll}
P ::= \overline{L}\ \{t\} & L ::= \texttt{class}\ C\ \texttt{extends}\ \overline{C}\ \{\overline{f}\ \overline{M}\} \\
M ::= m\ (\overline{x})\{t\} & t ::= v := \texttt{new}\ C() \mid v := p \mid v := e \\
p ::= e.m(\overline{e}) \mid m(\overline{e}) \mid m(\overline{e}) :> C & \quad \mid \texttt{skip} \mid \texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ t\ \texttt{fi} \mid t;t
\end{array}
$$

Fig. 1. The language syntax, in which C denotes a class name, m a method name, e an expression, and b a Boolean expression. Variables v are fields f or `return`. We use vector notation to denote lists and whitespace for the list concatenation operator (i.e., both $\overline{e}$ and $e\,\overline{e}$ are lists of expressions).

Late binding, or dynamic dispatch, is a central concept of object-orientation, already present in Simula [8]. A method call is late bound, or *virtual*, if the method body to be executed is selected at run-time, depending on the callee's actual class: if an object of class C_2 executes a method n defined in a superclass C_1 and this method calls method m defined in both classes, then the code selected for execution is associated to the *first matching occurrence* of m above C_2; i.e., m of C_2 is selected and not the one in C_1. If n, however, were executed in an instance of C_1, the virtual invocation of m would be bound to m's definition in C_1. We say that a definition of m is *reachable* from C if there is a class $D \leq C$ such that a call to m will bind to that definition for instances of D. For instance, if m is overridden by D, that definition is reached from C for instances of D. Thus, for a virtual call there might be several reachable definitions.

In contrast, multiple inheritance class hierarchies form acyclic directed graphs. This means that *horizontal* name conflicts may occur, as a class may have several direct superclasses. In the class hierarchy above a given class, several definitions of the same method may be reached, depending on the chosen path through the hierarchy. More elaborate binding strategies are needed to resolve horizontal conflicts. One solution is *explicit resolution*; e.g., to use qualification or renaming as in C++ [24], Eiffel [18], and POOL [2]. A second approach, which we follow in this paper, is to consider horizontal name conflicts as a natural feature of multiple inheritance. In particular when using libraries, the programmer cannot be expected to know (or resolve) potential name conflicts of, e.g., auxiliary methods in the libraries. Following [7, 9, 14], ambiguities can be solved by fixing the order in which inherited classes are searched; e.g., left to right.

2.1 A Multiple Inheritance Kernel Language

An object-oriented kernel language is given in Fig. 1, based on Featherweight Java [13]. A program P consists of a list $\overline{L}$ of class definitions and a method body. A class extends a list $\overline{C}$ of superclass names, which may be `Object`, with fields $\overline{f}$ and methods $\overline{M}$. A method M takes parameters $\overline{x}$ and contains a statement t. The sequential composition of statements t_1 and t_2 is written $t_1;t_2$. The statement $v := \texttt{new}\ C()$ creates a new object of class C with fields instantiated to default values, and assigns the new reference to v. In a method invocation $e.m(\overline{e})$, the object e receives a call to the method m with actual parameters $\overline{e}$. The expression $e.m(\overline{e})$ denotes a *virtual call*. (For convenience, we often write $e.m(\overline{e})$ or simply $e.m$ instead of $v := e.m(\overline{e})$.) The statement $m(\overline{e}) :> C$ denotes a *static call* to m above C; i.e., the call occurs in a class below C and it is bound above C independent of the actual class of the called object. This statement replaces the call to the superclass found in languages with single inheritance, C may

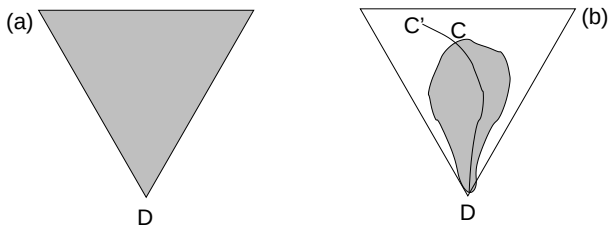

Fig. 2. Different binding strategies: (a) *maximal scope* and (b) *restricted scope*. For a given method m, C is the class making a call, D is the binding environment and C' is $bind(C,m)$, the default binding of m in C.

here be any superclass in the class hierarchy. Upon completion, a method activation returns the value of its `return` variable. Finally, there are standard statements for `skip`, conditionals `if` b `then` t `else` t `fi`, and assignments $v := e$. As usual, the pseudo-variable `this` is read-only. For simplicity, we assume given a language of side-effect free expressions e and we let fields have distinct names, methods with the same name have the same signature (i.e., no method overloading), class names be unique, programs be well-typed, and we ignore the types of fields and methods.

To make the representation of class hierarchies compact, a class name is bound to a tuple $\langle \overline{C}, \overline{f}, \overline{M} \rangle$ of type *Class*, where $\overline{C}$ is a list of superclass names, $\overline{f}$ a set of fields, and $\overline{M}$ a set of methods. The list $\overline{C}$ is assumed to consist of unique names and may be *nil*. The list of superclass names, field set, and method set of a class tuple are accessible by observer functions *inh*, *att*, and *mtds*, respectively.

2.2 The Binding of Virtual and Static Calls

We consider binding strategies for multiple inheritance class hierarchies with horizontal name conflicts. Let *Cid* and *Mid* denote types for class and method names. A call to a method m is bound with respect to a *search class* D; i.e., the search for a definition of m starts in D. Except for static calls, the search class is the callee's actual class.

In order to successfully bind a call to m, the name conflicts must be resolved: the binding algorithm traverses the class hierarchy in, e.g., a left-first depth-first manner. Assume that $D_1, \ldots, D_k$ are the direct superclasses of D. If m is not defined in D, search recursively for a definition of m in the class hierarchy above D_1. If a definition is not found above D_1, proceed to the next superclass D_2 of D. By type-safety, we may assume that a method definition is eventually found. This binding strategy is illustrated in Fig. 2(a), and defined by a partial function *bind*.

Definition 1. *Define* $bind : List[Cid] \times Mid \to Cid$ *as follows:*

$$\begin{array}{lll} bind(nil, m) & \triangleq & \bot \\ bind(C\ L, m) & \triangleq & C \qquad\qquad \text{if } m \in C.mtds \\ bind(C\ L, m) & \triangleq & bind(C.inh\ L, m) \quad \text{otherwise} \end{array}$$

For a virtual call to m on an instance of D, $bind(D,m)$ is the *default binding* of the call. Remark that a static call $m :> B$ is bound by $bind(B,m)$, regardless of the object's

actual class. With $bind(D,m)$, all local virtual calls to a method m in an instance of D are bound to the same definition, independent of where in the class hierarchy above D the call was initiated, as illustrated by the shaded area of Fig. 2(a).

This binding strategy above resolves horizontal as well as vertical name conflicts at the syntactic level, but may result in unexpected behavior. When two class hierarchies are merged in a common subclass, virtual calls in one hierarchy may suddenly be bound in the other, caused by unforeseen name conflicts. In order to extend the class hierarchy with a class D in a controlled way, the definition of every method m in $bind(D,m)$ must meet the requirements imposed by all calls to m above D. This is necessary even if m is not defined in D, which generally means that all calls made by some class above D must be considered. Thus, reasoning about program behavior quickly becomes intractable.

In order to better control method binding, we adapt the so-called *pruned binding strategy* [14]; whenever m is called from a method defined in class C, the call will, for instances of any subclass of C, be bound *below* the default binding $bind(C,m)$. (We shall refer to C as the calling class of m.) Intuitively, a call will only be bound to a definition which explicitly redefines the method definition known to the programmer. Technically, pruned binding is defined as follows:

Definition 2 (Pruned binding). *Define* $pbind : List[Cid] \times Cid \times Mid \rightarrow Cid$ *by:*

$$
\begin{array}{lll}
pbind(\ nil, C, m) \triangleq \bot & & \\
pbind(C\ L, C, m) \triangleq bind(C, m) & & \\
pbind(D\ L, C, m) \triangleq D & & \text{if } D < C \wedge m \in D.mtds \\
pbind(D\ L, C, m) \triangleq pbind(D.inh, C, m) & & \text{if } D < C \wedge m \notin D.mtds \\
pbind(D\ L, C, m) \triangleq pbind(L, C, m) & & \text{otherwise}
\end{array}
$$

Thus, $pbind(D,C,m)$ searches the class hierarchy above the search class D for a definition of m in a left-first, depth-first manner. In contrast to $bind(D,m)$, the search space is restricted by the calling class C. If the search reaches C, the default binding $bind(C,m)$ will be returned. Thus, the search is limited to the shaded area in Fig. 2(b). A virtual, local call to m inside C is bound by $pbind(D,C,m)$. Static analysis ensures the definedness of $bind(C,m)$, and thereby also of $pbind(D,C,m)$.

3 Lazy Behavioral Subtyping: Tracking Behavioral Constraints

Incremental reasoning means that extending a given class hierarchy by new classes does not require reverification of already established properties. Clearly, with late bound methods and without any restriction on how a class hierarchy can be extended, this goal cannot be achieved. The simple reason is that a virtual call to a method m refers to code which is not statically determined at compile-time (or verification-time). In an open setting the implementation referred to in the call to m may change, when m is overridden in a newly added class. Without imposing any restrictions, m's new implementation may be completely unrelated to the original one. Consequently, any correctness proof for the calling method which relies on properties of (the original) m is worthless and must be redone in the extended hierarchy. In summary, if the behavior of called methods is

allowed to change arbitrarily by inheritance (single inheritance or not), old proofs about code using those methods are invalidated when the methods' behavior changes.

That clearly indicates also how to get a grip on the problem: *restrict* the way a method's behavior may change by overriding. A classical way to do so is *behavioral subtyping* and works as follows. Assuming that the behavior of a method is logically specified by its pre- and postcondition (written $m(\overline{x}) : (p,q)\{t\}$), that specification "freezes" the behavior; i.e., it fixes the behavior for all later implementations. Behavioral subtyping supports incremental reasoning, but is too restrictive in practice [23]. *Lazy behavioral subtyping* [10] relaxes the regime regulating which properties need to be preserved: to avoid re-verification, it suffices to preserve properties that are *needed* when a method is used, not those announced by a method definition in its pre- and postcondition. Thus, this approach distinguishes the *commitment* of a method (the pre- and postcondition of a method definition), and the requirements to a method: a call-site to a method m, written $\{r\}\,x.m(e)\,\{s\}$ for a virtual call imposes the specification as a *requirement* to (reachable) method definitions for m.

The proof method for lazy behavioral subtyping has two parts. A conventional program logic (e.g., [3, 12, 19, 20]) and, on top of that, a framework which tracks commitments and requirements to methods as the program analysis incrementally proceeds.

Proof outlines. The proof system is presented using Hoare triples $\{p\}\,t\,\{q\}$, where p is the pre- and q the postcondition to the statement t. The meaning of a triple $\{p\}\,t\,\{q\}$ is that if t starts execution in a state where p holds and t terminates, then q holds after t. Triples can be derived in any suited program logic, so let $\vdash_{\mathrm{PL}} \{p\}\,t\,\{q\}$ denote that the triple $\{p\}\,t\,\{q\}$ is derivable in the chosen program logic PL. A *proof outline* [19] for a method definition $m(\overline{x})\{t\}$ is an annotated method $m(\overline{x}) : (p,q)\{t\}$ where method calls inside t are decorated with call-site requirements. We henceforth assume that all method bodies are decorated in this way. The derivability $\vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\{t\}$ of a proof outline is given by $\vdash_{\mathrm{PL}} \{p\}\,t\,\{q\}$. If p is the pre- and q the postcondition to some method (body), we call (p,q) a *specification* of that method (body).

Lazy behavioral subtyping for single inheritance. The core of the method is to appropriately track and manipulate behavioral constraints during the program analysis. This is done in a so-called *proof environment*. The proof environment in particular contains the constraints collected during analysis so far in the form of two mappings S and R. Given a class and a method name, the two mappings specify the associated commitments and requirements as sets of pre- and postcondition pairs. They are first explained for the single inheritance proof system.

Assume that we have verified a proof outline $m(\overline{x}) : (p,q)\{t\}$ in some class C. If m is *directly* defined in C (as opposed to inherited), the specification (p,q) is added to $S(C,m)$. If m is inherited and not redefined, m may guarantee different commitments in the context of C than those provided in the superclass which contains m's definition. In the last case, $S(C,m)$ acts as a *local extension* of the superclass commitments of m, containing the commitments known to be valid in C.

The analysis of a proof outline $m(\overline{x}) : (p,q)\{t\}$ in C imposes *requirements* to the methods called by m. For a call to n, the required specification (r,s) is given by an occurrence of a Hoare triple $\{r\}\,n\,\{s\}$ in the proof outline. Two steps are taken for

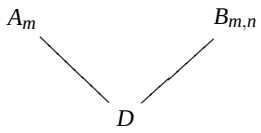

Fig. 3. A small multiple inheritance class hierarchy, focusing on definitions of methods *m* and *n*. The subscripts indicate the existence of a definition in the associated class.

each such requirement: The requirement is analyzed with regard to the definition of *n* reached for instances of *C*, and it is *remembered* in $R(C,n)$. The first step ensures that (r,s) is valid when *n* is executed by instances of *C*. To ensure that (p,q) remains a valid commitment when *m* is executed by an instance of a subclass of *C*, the second step imposes constraints on overridings of *n*: When *n* is overridden in a subclass of *C*, the requirements captured by $R(C,n)$ must hold for the new definition.

Additionally, the *S* mapping is used during the verification process to avoid reverification of specifications. Let $S{\uparrow}(C,n)$ be the union of $S(B,n)$ for all *B* between *C* and the first class above *C* that defines *n*. The verification of a requirement (r,s) succeeds immediately if can be concluded from $S{\uparrow}(C,n)$, denoted $S{\uparrow}(C,n) \twoheadrightarrow (r,s)$, where $\twoheadrightarrow$ is the entailment relation for specifications (see Appendix. A). Otherwise, an additional proof outline for (r,s) is needed. The successful analysis of this new proof outline will add (r,s) to $S(C,n)$, and it may cause new requirements on methods called by *n* in the context of *C*.

3.1 The Commitment Mapping for Multiple Inheritance

For single inheritance the binding of a virtual call to *m* is unambiguous wrt. a given search class *D*, as there is a unique path from *D* to the first class above *D* that implements *m*. Thus it is also unambiguous which method definition a given local extension of the commitment mapping refers to. This becomes ambiguous for multiple inheritance when the calling class is used for method binding, as illustrated by the following example.

Example 1 (Multiple inheritance). Consider the hierarchy in Fig. 3, where a method *m* is defined in classes *A* and *B* and not overridden in *D*. So $S(A,m)$ and $S(B,m)$ contain the commitments of *m* in *A* and *B*, respectively. Assume next that some method defined in *D* and calling *m* (via a self-call) is analyzed. This call is bound to *m* in *A*, starting the search in *D*. If the requirement of the call leads to the analysis of a new proof outline, the verified commitment is captured by the local extension $S(D,m)$. However, assume that later analysis of *D* requires to verify *n* of *B*, and that *m* is called from this method, as well. That call will be bound to *m* in *B*. With single inheritance, we could then rely on $S(D,m)$ when analyzing the requirement of this call. However, this is no longer safe, as $S(D,m)$ contains local commitments of *m as inherited from A*. Furthermore, if the call in *n* leads to an analysis of a new proof outline for *m* in *B*, a new commitment should be captured by a local extension for *D*. However, we cannot distinguish the local commitment extension for *m* in *A* from the one for *m* in *B*. □

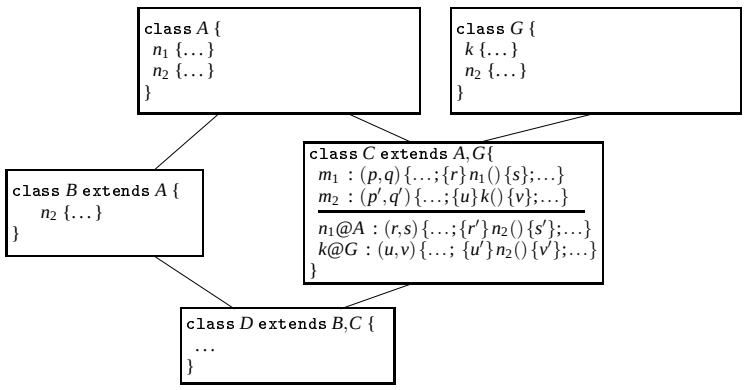

Fig. 4. An example class hierarchy. The notation $n@E : (p,q)\{t\}$ used below the line in class C indicates proof outlines of n defined in E generated during the analysis of C.

In order to adapt the commitments to multiple inheritance, we let the mapping take an additional class argument, such that $S(C,B,m)$ returns the commitments of m *as defined in* B, that are established during analysis of C.

Example 2. Referring to Fig. 3, the set $S(A,A,m)$ is built during the analysis of A, and similarly for B. Furthermore, $S(D,A,m)$ and $S(D,B,m)$ return the local extensions of m in A and B, respectively. During the analysis of a method defined in D, a call to m can rely on both $S(A,A,m)$ and $S(D,A,m)$, and $S(D,A,m)$ is extended if the call requires the analysis of a new proof outline for m in A. For class B, the analysis of the call to m from n in the context D, can rely on $S(B,B,m)$ and $S(D,B,m)$, and the set $S(D,B,m)$ is extended if a new proof outline for m in B is analyzed. □

The set $S(D,C,m)$ is only extended during analysis of D. Therefore, whenever $S(D,C,m)$ is non-empty, the class D is below C, and m is defined in C. The commitment collecting function $S{\uparrow}(D,C,m)$ is defined such that $S{\uparrow}(D,C,m)$ collects the union of $S(B,C,m)$ for all classes B such that $D \leq B \leq C$. The formal definition of this function can be found in Appendix A.

3.2 The Requirement Mapping for Multiple Inheritance

Next we consider the tracking of method requirements for multiple inheritance. We start by illustrating that the requirement mapping for single inheritance is too weak in the presence of multiple inheritance.

Example 3. Consider the diamond-structured hierarchy in Fig. 4. We focus on the analysis of class C. Let (p,q) be the commitment of m_1 of C as indicated by the figure, i.e., $S(C,C,m_1) = \{(p,q)\}$, which leads to a requirement (r,s) on n_1. Assume that (r,s) does not follow from the already established commitments of n_1 in A, such that a new proof outline is analyzed, as indicated below the line in C. After this analysis, we

have $R(C,n_1) = \{(r,s)\}$, $S(C,A,n_1) = \{(r,s)\}$, $R(C,n_2) = \{(r',s')\}$, and $S(C,A,n_2) = \{(r',s')\}$. Thus, the specifications (r,s) and (r',s') are established wrt. the definitions of n_1 and n_2 in A. For the analysis of B, the independence of B and C implies that requirement (r',s') is not imposed on n_2 in B when B is analyzed; B is not a subclass of C. This, however, leads to a potential soundness problem when D is introduced. Method m_1 is inherited from C to D. Nonetheless, a call to m_1 from D cannot rely on $S(C,C,m_1)$ unless (r',s') is verified for n_2 in B.

To ensure soundness, we therefore have to verify (r',s'), included in the set $R(C,n_2)$, with respect to n_2 in B when D is introduced. However, it is unnecessary restrictive to verify all elements of $R(C,n_2)$ against n_2 in B, as illustrated by class G in Fig. 4. Assume that a proof outline for $\{u\}\,k()\,\{v\}$ is analyzed during analysis of C. Since there is a call $\{u'\}\,n_2()\,\{v'\}$ in the proof outline of k, the set $R(C,n_2)$ will contain the element (u',v'). For an instance of D, the call to n_2 in G will still be bound to G, which means that the requirement (u',v') should not be imposed on n_2 in B. □

For a class C, let *commSup*(C) return the name of classes that are above *more* than one class in $C.inh$. We say that a diamond is introduced by C if *commSup*$(C) \neq \emptyset$. A class B is in *commSup*(C) if there are two different classes C_1 and C_2 in $C.inh$ such that $C_1 \leq B$ and $C_2 \leq B$, and we then refer to B as a *common superclass* of C. The above example illustrates that whenever C is analyzed, it might be necessary to verify some requirements made by classes between C and the common superclasses of C. The requirements that need verification are generated during analysis of a proof outline in a common superclass. To handle this problem, the multiple inheritance definition of the requirement mapping is more fine-grained than for single inheritance. For two classes C and D where $C \leq D$, we let $R(C,D,m)$ return the requirements towards m that are needed during verification of C for a calling class D. Thus, the calls to m occur syntactically in D. The more fine-grained requirement mapping is illustrated by the following example:

Example 4. Reconsider the class hierarchy in Ex. 3. For the analysis of m_1 in C, the requirement (r,s) is added to $R(C,C,n_1)$. Furthermore, the analysis of the proof outline for n_1 adds (r',s') to $R(C,A,n_2)$. For the analysis of m_2, we get $R(C,C,k) = \{(u,v)\}$ and $R(C,G,n_2) = \{(u',v')\}$. Consequently, we can distinguish (r',s') as required by A, from (u',v') as required by G. □

The commitments of a class C are captured by different $S(C,\_,\_)$ sets, and the requirements on which these commitments rely are captured by $R(C,\_,\_)$ sets. Lazy behavioral subtyping for multiple inheritance applies the following strategy: in order to ensure that the commitments of the methods of C are valid when these methods are executed in an instance of a subclass, the requirements imposed by C must hold when calls are bound with the subclass as search class. If m is overridden by a subclass G of C, all requirements on m made above G must be verified for the new definition. This is captured by a verification of the requirements returned by $R{\uparrow}(G,m)$, which denotes the union of all requirements to m made by classes above G. Consider next a requirement set $R(C,D,m)$ of C, and assume that m is not overridden by the subclass G of C. There is a special case where D is a common superclass of G. Calls made by D may then be bound in a class which is neither below nor above C; i.e., the requirements made by C was not imposed on this class when it was analyzed. The same argument applies to other

classes above G, and in general we let the set $dreq(G,D,m)$ of *diamond requirements* be the union of $R(C,D,m)$ for all such C between G and D. Verification of G leads to a verification of $dreq(G,D,m)$ with regard to the definition of m to which the calls from D are bound for instances of G. We refer to Appendix A for formal definitions of $dreq$ and $R\uparrow$.

Example 5. Referring to Ex. 3, the class A is a common superclass of D. The requirement (r',s') contained in $R(C,A,n_2)$, is returned by $dreq(D,A,n_2)$, and (r',s') is then verified with regard to n_2 in B. Remark that (u',v') in $R(C,G,n_2)$ is not in $dreq(D,A,n_2)$.

3.3 Proof Environments for Multiple Inheritance Class Hierarchies

We now give the formal definition of proof environments and their soundness.

Definition 3 (Proof environments for multiple inheritance). *A* proof environment $\mathcal{E}$ *of type Env is a triple* $\langle P,S,R\rangle$*, where* $P : Cid \to Class$ *is a partial mapping,* R *and* S *are total mappings of type* $Cid \times Cid \times Mid \to Set[Prop]$.

In an environment $\mathcal{E}$, P reflects the class structure, $S(C,D,m)$ the set of commitments of m defined in D with respect to a subclass C, and $R(C,D,m)$ the set of requirements to m from C that must be ensured by classes below C. If the proof environment of a mapping is not clear from the context, we use a subscript; e.g., $R_{\mathcal{E}}$.

Definition 4 (Sound environments). *A* sound environment $\mathcal{E}$ *satisfies the following conditions for all* $C,D \in \mathcal{E}$ *and* $m : Mid$*:*

$$\begin{aligned}
&\forall (p,q) \in S_{\mathcal{E}}(C,D,m)\ .\ \exists body_{\mathcal{E}}(D,m)\ .\ \vdash_{\mathrm{PL}} m(\overline{x})\ :\ (p,q)\,\{body_{\mathcal{E}}(D,m)\}\\
&\ \wedge\ \forall \{r\}\,n\,\{s\} \in body_{\mathcal{E}}(D,m)\ .\ \forall G \leq_{\mathcal{E}} C\ .\ S{\uparrow_{\mathcal{E}}}(G, pbind_{\mathcal{E}}(G,D,n),n) \twoheadrightarrow (r,s))\\
&\ \wedge\ \forall \{r\}\,n :> B\,\{s\} \in body_{\mathcal{E}}(D,m)\ .\ S{\uparrow_{\mathcal{E}}}(C, bind_{\mathcal{E}}(B,n),n) \twoheadrightarrow (r,s)\\
&\ \wedge\ \forall \{r\}\,x.n\,\{s\} \in body_{\mathcal{E}}(D,m)\ .\ \exists E\ .\ ((x:E) \in {\uparrow}C.att) \Rightarrow S{\uparrow_{\mathcal{E}}}(E,n) \twoheadrightarrow (r,s)
\end{aligned}$$

Informally, a proof environment $\mathcal{E}$ is *sound* if, whenever $(p,q) \in S_{\mathcal{E}}(C,D,m)$, there is a proof outline for m in D with respect to the commitment (p,q). Furthermore, for each requirement $\{r\}\,n\,\{s\}$ in this proof outline and each subclass G of C, (r,s) must follow from the commitments of the method to which a call is bound for search class G. For each external call $\{r\}\,x.n\,\{s\}$, the requirement must follow from the commitments of n in D. Let $S{\uparrow_{\mathcal{E}}}(C,m)$ abbreviate $S{\uparrow_{\mathcal{E}}}(C, bind_{\mathcal{E}}(C,m),m)$, $body_{\mathcal{E}}(D,m)$ return the body of m in D, and $C \in \mathcal{E}$ denote that $P_{\mathcal{E}}(C)$ is defined.

Remark that the requirement mapping is not visible in Def. 4. However, it is needed to show that the calculus maintains environment soundness. Let $\models_C \{p\}\,t\,\{q\}$ denote $\models \{p\}\,t\,\{q\}$ under the assumption that virtual calls in t are bound in the context of C, and let $\models_C m(\overline{x})\ :\ (p,q)\,\{t\}$ be given by $\models_C \{p\}\,t\,\{q\}$. If there are no method calls in t and $\vdash_{\mathrm{PL}} \{p\}\,t\,\{q\}$, then $\models \{p\}\,t\,\{q\}$ follows by the soundness of PL. Lemma 1 below states that if $(p,q) \in S_{\mathcal{E}}(C,D,m)$ and a method m in D is executed in an instance of a subclass of C, a sound environment guarantees that (p,q) is a valid commitment:

Lemma 1. *Given a sound environment* $\mathcal{E}$ *and a sound program logic* PL*. For all* $C,D :$ *Cid,* $m :$ *Mid, and* $(p,q) :$ *Prop such that* $C,D \in \mathcal{E}$ *and* $(p,q) \in S{\uparrow_{\mathcal{E}}}(C,D,m)$*, we have* $\models_C m(\overline{x})\ :\ (p,q)\,\{body_{\mathcal{E}}(D,m)\}$.

Proof. By induction on the call structure of m. Since $(p,q) \in S{\uparrow_{\mathcal{E}}}(C,D,m)$, there must, by Def. 4, exist some class B such that $C \leq_{\mathcal{E}} B \leq_{\mathcal{E}} D$ and $(p,q) \in S_{\mathcal{E}}(B,D,m)$. In addition, there must exist a proof outline $body_{\mathcal{E}}(D,m)$ for the method such that $\vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\{body_{\mathcal{E}}(D,m)\}$.

Base case: The execution $\{p\}\, body_{\mathcal{E}}(D,m)\, \{q\}$ does not lead to any method calls. Then $\models_C m(\overline{x}) : (p,q)\{body_{\mathcal{E}}(D,m)\}$ follows by the soundness of PL.

Induction step: Consider a method call $\{r\}\, n\, \{s\}$ in the method $body_{\mathcal{E}}(D,m)$, and let $H = pbind_{\mathcal{E}}(C,D,n)$. Assume $\models_C n(\overline{y}) : (p',q')\{body_{\mathcal{E}}(H,n)\}$ as induction hypothesis for each $(p',q') \in S{\uparrow_{\mathcal{E}}}(C,H,n)$. Consequently, it suffices to ensure $S{\uparrow_{\mathcal{E}}}(C,H,n) \twoheadrightarrow (r,s)$, which follows by Def. 4. A corresponding argument applies to $\{r\}\, n :> B\{s\}$.

4 The Inference System

Based on the proof environments, we next present the derivation system, given as a set of inference rules, analyzing and manipulating the proof environment.

4.1 Analysis Operations

In the calculus, judgments have the form $\mathcal{E} \vdash \mathcal{A}$, where $\mathcal{E}$ is the proof environment and $\mathcal{A}$ is a list of *analysis operations* with the following syntax.

$$
\begin{array}{lll}
O ::= \varepsilon \mid \mathit{analyzeMtds}(\overline{M}) \mid \mathit{verify}(C,m,\overline{R}) \mid \mathit{analyzeOutline}(C,t) & & \text{single class} \\
\quad \mid \mathit{supCls}(\overline{C}) \mid \mathit{supMtd}(C,\overline{m}) \mid O \cdot O & & \\
\mathcal{S} ::= \emptyset \mid L \mid \mathit{require}(C,m,(p,q)) \mid \mathcal{S} \cup \mathcal{S} & & \text{set of classes} \\
\mathcal{A} ::= \mathit{module}(\overline{L}) \mid [\langle C : O\rangle ; \mathcal{S}] \mid [\varepsilon ; \mathcal{S}] \mid \mathit{module}(\overline{L}) \cdot \mathcal{A} & & \text{seq. of modules}
\end{array}
$$

The rule system below roughly specifies an algorithm that recursively traverses a class hierarchy and its syntactic constituents — classes, methods, statements, etc. — according to the principles explained in Section 3; in particular, tracking commitments and requirements. A program is given as a sequence of modules, where a module is a set of classes considered as a *compilation unit*. Programs are open in the sense that at later stages, the class hierarchy may be extended. However, at each stage of the development, the modules given so far represent a complete, compilable program. Hence, modules are analyzed in sequential order, whereas classes inside a module are simply represented as a set. The operations above, together with a proof environment, steer the algorithm through the program (which is assumed to be syntactically well-formed and well-typed). The analysis starts with an $\mathcal{E} \vdash \mathcal{A}$ where $\mathcal{E}$ is empty and $\mathcal{A}$ contains the program as a sequence of modules. On the level of classes, the set $\mathcal{S}$ contains a module's classes. However, the inference rules ensure that a class can only be analyzed after all its superclasses have been analyzed. The operation $\texttt{class}\, C\, \texttt{extends}\, \overline{D}\, \{\overline{f}\, \overline{M}\}$ initiates the analysis of C, and $[\langle C : O\rangle ; \mathcal{S}]$ analyzes O in the context of class C *before* operations in $\mathcal{S}$ are considered. The analysis of a specific class involves the analysis the proof outlines for its methods $\overline{M}$, the verification of the requirements for a method, and collecting the proof obligations for the calls mentioned inside the method bodies (by the operations $\mathit{analyzeMtds}(\overline{M})$, $\mathit{verify}(D,m,\overline{R})$, and $\mathit{analyzeOutline}(D,t)$). The operation

require$(D,m,(p,q))$ applies to external calls to ensure that m in D satisfies the requirement (p,q). Requirements are lifted outside the context of the analyzed class by this operation, and the verification of requirement (p,q) for m in D is shifted into the set $\mathcal{S}$ of analysis operations. The remaining two operations, *supCls*$(\overline{D})$ and *supMtd*$(D,\overline{m})$ are used during analysis of C, if C introduces diamonds in the class hierarchy. The operation *supCls*$(\overline{D})$ takes a list of class names and generates a *supMtd*$(D,\overline{m})$ for each $D \in \overline{D}$ where $\overline{m}$ are the names of the methods that are called by D.

Environment updates are represented by the operator $\_ \oplus \_ : \mathit{Env} \times \mathit{Update} \to \mathit{Env}$, where the second argument represents the update. There are three different environment updates; loading a new class and extending the commitments or the requirements of a method in a class. The updates are defined as follows:

$$\mathcal{E} \oplus \texttt{class}\ C\ \texttt{extends}\ \overline{D}\ \{\overline{f}\ \overline{M}\} = \langle P_{\mathcal{E}}[C \mapsto \langle \overline{D}, \overline{f}, \overline{M}\rangle], S_{\mathcal{E}}, R_{\mathcal{E}}\rangle$$
$$\mathcal{E} \oplus \mathit{extS}(C,D,m,(p,q)) = \langle P_{\mathcal{E}}, S_{\mathcal{E}}[(C,D,m) \mapsto S_{\mathcal{E}}(C,D,m) \cup \{(p,q)\}], R_{\mathcal{E}}\rangle$$
$$\mathcal{E} \oplus \mathit{extR}(C,D,m,(p,q)) = \langle P_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}}[(C,D,m) \mapsto R_{\mathcal{E}}(C,D,m) \cup \{(p,q)\}]\rangle$$

4.2 The Inference Rules

The inference rules are given in Fig. 5 and 6. Rule (NewModule) initiates the analysis of a set of classes. Furthermore, (NewClass) loads a new class C for analysis, the second premise ensures that the superclasses $\overline{D}$ have already been analyzed. For each method m in C, the calculus generates an operation *verify*$(C,m,\overline{R})$, where $\overline{R}$ is the set of requirements that must hold for this method. Rules (ReqDer) and (ReqNotDer) deal with the verification of a particular property with respect to the implementation. If the property follows from the established specification of the method, rule (ReqDer) continues with the remaining analysis operations. Otherwise, a proof of the property is required. By (ReqNotDer), an outline of the method property is then analyzed by an *analyzeOutline* operation. Remark that (ReqNotDer) is the only rule which extends the S mapping.

The rule (Call) analyzes a requirement to a virtual call occurring in some proof outline. The rule leads extends the R mapping and generates a *verify* operation to analyze the requirement for the implementation to which the call will bind. The extension of R ensures that future redefinitions of m respect the requirement; i.e., when a new implementation is considered by (NewMtd). Rule (SupCall) also generates a *verify* operation, but does not extend R. External calls are handled by the rules (ExtReq) and (ExtCall).

Fig. 6 contains rules for analyzing requirements from common superclasses when *diamonds* are introduced in the environment. Rule (SupMtd) generates a *supMtd* for each common superclass. For each of these superclasses, (SupReq) generates a *verify* operation for each method called by the class. If a class C is introduced by (NewClass) where C does not have any common superclasses, the *supCls* operation generated by the rule will have an empty argument. This operation is then discarded by (NoSup). Some structural rules are left out from Fig. 5 and Fig. 6. These can be found in Appendix C.

Theorem 1. *Let $\mathcal{E}$ be a sound environment and $\overline{L}$ a set of class declarations. If a proof of $\mathcal{E} \vdash \mathit{module}(\overline{L})$ has $\mathcal{E}'$ as its resulting environment, then $\mathcal{E}'$ is also sound.*

$$\frac{\mathcal{E} \vdash [\varepsilon; \overline{L}] \cdot \mathcal{A}}{\mathcal{E} \vdash \mathit{module}(\overline{L}) \cdot \mathcal{A}} \text{ (NewModule)}$$

$$\frac{C \notin \mathcal{E} \qquad \overline{D} \neq \mathit{nil} \Rightarrow \overline{D} \in \mathcal{E} \qquad \overline{E} = \mathit{commSup}_{\mathcal{E}}(C) \quad \mathcal{E} \oplus (\texttt{class}\ C\ \texttt{extends}\ \overline{D}\ \{\overline{f}\ \overline{M}\}) \vdash [\langle C : \mathit{analyzeMtds}(\overline{M}) \cdot \mathit{supCls}(\overline{E})\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\varepsilon; \texttt{class}\ C\ \texttt{extends}\ \overline{D}\ \{\overline{f}\ \overline{M}\}\ \mathcal{S}] \cdot \mathcal{A}} \text{ (NewClass)}$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{verify}(C, m, (p,q) \cup R\uparrow_{\mathcal{E}} (C.\mathit{inh}, m)) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeMtds}(m(\overline{x}) : (p,q)\,\{t\}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (NewMtd)}$$

$$\frac{S\uparrow_{\mathcal{E}}(C,D,m) \twoheadrightarrow (p,q) \qquad \mathcal{E} \vdash [\langle C : O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{verify}(D, m, (p,q)) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (ReqDer)}$$

$$\frac{\vdash_{\mathrm{PL}} m : (p,q)\,\{\mathit{body}_{\mathcal{E}}(D,m)\} \quad \mathcal{E} \oplus \mathit{extS}(C,D,m,(p,q)) \vdash [\langle C : \mathit{analyzeOutline}(D, \mathit{body}_{\mathcal{E}}(D,m)) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{verify}(D, m, (p,q)) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (ReqNotDer)}$$

$$\frac{\mathit{pbind}_{\mathcal{E}}(C,D,m) = E \quad \mathcal{E} \oplus \mathit{extR}(C,D,m,(p,q)) \vdash [\langle C : \mathit{verify}(E, m, (p,q)) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeOutline}(D, \{p\}\, m\, \{q\}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (Call)}$$

$$\frac{\mathit{bind}_{\mathcal{E}}(B,m) = A \qquad \mathcal{E} \vdash [\langle C : \mathit{verify}(A, m, (p,q)) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeOutline}(D, \{p\}\, m :> B\, \{q\}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (SupCall)}$$

$$\frac{x : E \in \uparrow C.\mathit{att} \qquad \mathcal{E} \vdash [\langle C : O\rangle ; \mathcal{S} \cup \mathit{require}(E, m, (p,q))] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeOutline}(D, \{p\}\, x.m\, \{q\}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (ExtCall)}$$

$$\frac{C \in \mathcal{E} \qquad D = \mathit{bind}_{\mathcal{E}}(C,m) \qquad S\uparrow_{\mathcal{E}}(C,D,m) \twoheadrightarrow (p,q) \qquad \mathcal{E} \vdash [\varepsilon; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\varepsilon; \mathit{require}(C, m, (p,q)) \cup \mathcal{S}] \cdot \mathcal{A}} \text{ (ExtReq)}$$

$$\frac{\mathcal{E} \vdash [\varepsilon; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \varepsilon\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (EmpClass)} \qquad \frac{\mathcal{E} \vdash \mathcal{A}}{\mathcal{E} \vdash [\varepsilon; \emptyset] \cdot \mathcal{A}} \text{ (EmpPack)}$$

Fig. 5. The inference system. Here m denotes a call, including actual parameters.

Proof. The auxiliary lemmas used in this proof can be found in Appendix. B. Assume given a sound environment $\mathcal{E}$. The proof is by induction over the inference rules. The only rule that extends $S_{\mathcal{E}}(C,D,m)$ is (ReqNotDer), and this rule ensures that there is a proof outline $\mathit{body}_{\mathcal{E}}(D,m)$ of m defined in D such that $\vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\,\{\mathit{body}_{\mathcal{E}}(D,m)\}$ for each $(p,q) \in S_{\mathcal{E}}(C,D,m)$ and we must have $C \leq_{\mathcal{E}} D$.

For each $\{r\}\, n\, \{s\}$ in the proof outline we then have $n \in \mathit{called}_{\mathcal{E}}(D)$ and the rule (Call) will ensure $R_{\mathcal{E}}(C,D,n) \twoheadrightarrow (r,s)$. As the class hierarchy evolves, we then need to ensure $S\uparrow_{\mathcal{E}}(G, \mathit{pbind}_{\mathcal{E}}(G,D,n), n) \twoheadrightarrow (r,s)$ for all classes $G \leq_{\mathcal{E}} C$. This is done by induction over the depth d of the class hierarchy below C.

Base case: $d = 0$, i.e., $G = C$. In this case, we need to ensure $S\uparrow_{\mathcal{E}}(C,H,n) \twoheadrightarrow (r,s)$ for $H = \mathit{pbind}_{\mathcal{E}}(C,D,n)$. When the rule (ReqNotDer) is applied, including (p,q) in $S_{\mathcal{E}}(C,D,m)$, an operation $\mathit{analyzeOutline}(D, \{r\}\, n\, \{s\})$ is generated and analyzed in the context of C. The rule (Call) will then lead to an operation $\mathit{verify}(H, n, (r,s))$.

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{supMtd}(D, \mathit{called}_{\mathcal{E}}(D) \setminus C.\mathit{mtds}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{supCls}(D) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (SUPMTD)}$$

$$\frac{E = \mathit{pbind}_{\mathcal{E}}(C,D,m) \qquad \mathcal{E} \vdash [\langle C : \mathit{verify}(E,m,\mathit{dreq}(C,D,m)) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{supMtd}(D,m) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (SUPREQ)}$$

$$\frac{\mathcal{E} \vdash [\langle C : O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{supCls}(\emptyset) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (NOSUP)}$$

$$\frac{\mathcal{E} \vdash [\langle C : O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{supMtd}(D,\emptyset) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (NOSUPMTD)}$$

Fig. 6. The extension of the inference system with rules for analyzing of requirements made by common superclasses.

Since the operation succeeds, either rule (REQDER) or (REQNOTDER) is applied. The relation $S{\uparrow_{\mathcal{E}}}(C,H,n) \twoheadrightarrow (r,s)$ must hold directly if (REQDER) is applied. Otherwise, if (REQNOTDER) is applied, the set $S_{\mathcal{E}}(C,H,n)$ is extended with (r,s). The desired relation then holds since $S{\uparrow_{\mathcal{E}}}(C,H,n) \twoheadrightarrow S_{\mathcal{E}}(C,H,n)$.

Induction Step: $d = d' + 1$, i.e., $G <_{\mathcal{E}} C$ at depth d below C. For all classes G' at depth d' such that $G' \leq_{\mathcal{E}} C$, we assume $S{\uparrow_{\mathcal{E}}}(G', \mathit{pbind}_{\mathcal{E}}(G',D,n),n) \twoheadrightarrow (r,s)$ as the induction hypothesis.

We consider two cases: $n \in G.\mathit{mtds}$ and $n \notin G.\mathit{mtds}$.

Case 1: $n \in G.\mathit{mtds}$. Since $\mathit{pbind}_{\mathcal{E}}(G,D,n) = G$, the relation $S_{\mathcal{E}}(G,G,n) \twoheadrightarrow (r,s)$ must be ensured. By Def. 6, we have $R{\uparrow_{\mathcal{E}}}(G,n) \twoheadrightarrow R_{\mathcal{E}}(C,D,n)$ since $G <_{\mathcal{E}} C \leq_{\mathcal{E}} D$. Therefore, the rule (NEWMTD) will initiate an operation $\mathit{verify}(G,n,(r,s))$ which is analyzed in the context of G. This operation either succeeds by (REQDER) or (REQNOTDER), both ensuring the desired $S_{\mathcal{E}}(G,G,n) \twoheadrightarrow (r,s)$.

Case 2: $n \notin G.\mathit{mtds}$. Let $E = \mathit{lm}(G.\mathit{inh},D)$ (see Def. 8). Thus, E is the leftmost class in $G.\mathit{inh}$ that is below D. Furthermore, let $H = \mathit{pbind}_{\mathcal{E}}(E,D,n)$. We here distinguish between two cases, depending on whether E is below C or not.

Case 2a: $E \leq_{\mathcal{E}} C$. Since $E \in G.\mathit{inh}$ and $E \leq_{\mathcal{E}} C$, we know that E is at depth d' below C, and may use the induction hypothesis to assume $S{\uparrow_{\mathcal{E}}}(E,H,n) \twoheadrightarrow (r,s)$. By Lemma 2, we have $\mathit{pbind}_{\mathcal{E}}(G,D,n) = \mathit{pbind}_{\mathcal{E}}(E,D,m)$ which gives $\mathit{pbind}_{\mathcal{E}}(G,D,n) = H$. The desired relation $S{\uparrow_{\mathcal{E}}}(G,H,n) \twoheadrightarrow (r,s)$ then follows since $S{\uparrow_{\mathcal{E}}}(E,H,n) \subseteq S{\uparrow_{\mathcal{E}}}(G,H,n)$ by Def. 5.

Case 2b: $E \not\leq_{\mathcal{E}} C$. Since $E \leq_{\mathcal{E}} D$, we must have $C \neq D$. Since $G <_{\mathcal{E}} C$, there must exist some $E' = \mathit{lm}(G.\mathit{inh},C)$. Since $E \not\leq_{\mathcal{E}} C$, we must have $E \neq E'$. By $E \leq_{\mathcal{E}} D$ and $E' \leq_{\mathcal{E}} C <_{\mathcal{E}} D$, we then know that D is a common superclass of G, i.e., $D \in \mathit{commSup}_{\mathcal{E}}(G)$. Analysis of G will initiate a $\mathit{supCls}(\mathit{commSup}_{\mathcal{E}}(G))$ operation. Application of rule (DECOMPSUP) will generate a $\mathit{supCls}(D)$ operation. Since $n \in \mathit{called}_{\mathcal{E}}(D) \setminus G.\mathit{mtds}$, rules (SUPMTD) and (DECOMPSUPMTD) will generate a $\mathit{supMtd}(D,n)$ operation. Rule (SUPREQ) will then initiate a $\mathit{verify}(H,n,\mathit{dreq}(G,D,n))$ operation. Analysis of each requirement in this set either succeeds by application of (REQDER) or (REQNOTDER), which ensures $S{\uparrow_{\mathcal{E}}}(G,H,n) \twoheadrightarrow \mathit{dreq}(G,D,n)$. The desired conclusion $S{\uparrow_{\mathcal{E}}}(G,H,n) \twoheadrightarrow (r,s)$ fol-

lows by transitivity of $\twoheadrightarrow$ since $R_{\mathcal{E}}(C,D,n) \twoheadrightarrow (r,s)$, and $R_{\mathcal{E}}(C,D,n) \subseteq \mathit{dreq}(G,D,n)$ by Lemma 3.

5 Related Work

Multiple inheritance is supported in, e.g., C++ [24], CLOS [9], Eiffel [18], POOL [2], and Self [7]. Horizontal name conflicts in C++, POOL, and Eiffel are removed by explicit resolution, after which the inheritance graph may be linearized. Multiple dispatch, or multi-methods [9], gives a more powerful binding mechanism, but reasoning about multi-methods and redefinition is difficult. The prototype-based language Self [7] proposes an elegant *prioritized binding strategy*. Each superclass is given a priority. With equal priority, the superclass related to the caller class is preferred. However, explicit class priorities may cause surprises in large class hierarchies: names may become ambiguous through inheritance. If neither class is related to the caller, binding fails.

Formalizations of multiple inheritance in the literature traditionally use the *objects-as-records* paradigm. This approach addresses subtyping issues related to subclassing, but method binding is not easily captured. In Cardelli's denotational semantics of multiple inheritance [6], not even access to methods of superclasses is addressed. Rossie, Friedman, and Wand [22] formalize multiple inheritance using *subobjects*, a run-time data structure used for virtual pointer tables [15,24]. This work focuses on compile-time issues and does not clarify multiple inheritance at the abstraction level of the programming language. A natural semantics for virtual binding in Eiffel models the binding mechanism at the abstraction level of the program [4]. Recently, an operational semantics and type safety proof inspired by C++ has been formalized in Isabelle [25].

Work on behavioral reasoning about object-oriented programs address languages with single inheritance (e.g., [5, 20, 21]). For late binding, different variations of behavioral subtyping are most common [1,16,17], as discussed above. Pierik and de Boer [20] present a sound and complete reasoning system for late bound calls which does not rely on behavioral subtyping. This work, also for single inheritance, is based on a closed world assumption, meaning that the class hierarchy is not open for incremental extensions. To support object-oriented design, proof systems should be constructed for incremental reasoning. We are not aware of proof systems for multiple inheritance.

6 Conclusion

Lazy behavioral subtyping supports incremental reasoning under an open world assumption, where class hierarchies can be extended by inheritance. The approach is more flexible than traditional behavioral subtyping. In this paper, we have extended the lazy behavioral subtyping approach to a language with multiple inheritance, based on a pruned binding strategy for virtual calls. The combination of pruned binding and lazy behavioral subtyping has the advantage that requirements from two independent class hierarchies do not interfere with each other when the hierarchies are combined in a common subclass. This is essential in an incremental proof system. The inference rules for incremental reasoning are essentially syntax-driven and would form a good basis for combining behavioral reasoning in a program development environment.

References

1. P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 60–90. Springer, 1991.
2. P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 25(10), pages 161–168. ACM Press, Oct. 1990.
3. K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Systems*. Texts and Monographs in Computer Science. Springer, 1991.
4. I. Attali, D. Caromel, and S. O. Ehmety. A natural semantics for Eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems*, 18(6):711–729, 1996.
5. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
6. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2-3):138–164, 1988.
7. C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation*, 4(3):207–222, 1991.
8. O.-J. Dahl, B. Myhrhaug, and K. Nygaard. (Simula 67) Common Base Language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, May 1968.
9. L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *European Conference on Object-Oriented Programming (ECOOP'87)*, volume 276 of *LNCS*, pages 151–170. Springer, 1987.
10. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. In J. Cuellar and T. Maibaum, editors, *Proc. 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *LNCS*, pages 52–67. Springer, May 2008. To Appear.
11. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the Join calculus. *Journal of Logic and Algebraic Programming*, 57(1-2):23–69, 2003.
12. C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.
13. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
14. E. B. Johnsen and O. Owe. A dynamic binding strategy for multiple inheritance and asynchronously communicating objects. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. 3rd International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *LNCS*, pages 274–295. Springer, 2005.
15. S. Krogdahl. Multiple inheritance in Simula-like languages. *BIT*, 25(2):318–326, 1985.
16. G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20a, Department of Computer Science, Iowa State University, Ames, Iowa, 2006.
17. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
18. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997.
19. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

20. C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 343(3):413–442, 2005.
21. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *8th European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.
22. J. G. Rossie Jr., D. P. Friedman, and M. Wand. Modeling subobject-based inheritance. In P. Cointe, editor, *10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *LNCS*, pages 248–274. Springer, July 1996.
23. N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. Fifth International Conference on Software Reuse (ICSR5)*, pages 206–215. IEEE Computer Society Press, 1998.
24. B. Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–395, Dec. 1989.
25. D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA'06)*, pages 345–362. ACM, 2006.

A Auxiliary Functions

In the following definitions, we assume $m : Mid$, $C, D, E : Cid$, and $L : List[Cid]$.

Definition 5 (Commitment collection). *Define* $S\uparrow : List[Cid] \times Cid \times Mid \to Set[Prop]$*:*

$$\begin{array}{lll} S\uparrow(nil, C, m) & \triangleq & \emptyset \\ S\uparrow(C\ L, C, m) & \triangleq & S(C, C, m) \cup S\uparrow(L, C, m) \\ S\uparrow(D\ L, C, m) & \triangleq & S(D, C, m) \cup S\uparrow(D.inh\ L, C, m) \quad \text{if } D < C \\ S\uparrow(D\ L, C, m) & \triangleq & S\uparrow(L, C, m) \quad \text{otherwise} \end{array}$$

Definition 6 (Requirement collection). *Define* $R\uparrow : List[Cid] \times Mid \to Set[Prop]$ *by:*

$$\begin{array}{lll} R\uparrow(nil, m) & \triangleq & \emptyset \\ R\uparrow(C\ L, m) & \triangleq & R(C, m) \cup R\uparrow(C.inh\ L, m) \end{array}$$

where $R : Cid \times Mid \to Set[Prop]$ *is defined by*

$$R(C, m) \triangleq req(C, C, m)$$

and $req : Cid \times List[Cid] \times Mid \to Set[Prop]$ *is defined by:*

$$\begin{array}{lll} req(C, nil, m) & \triangleq & \emptyset \\ req(C, D\ L, m) & \triangleq & R(C, D, m) \cup req(C, D.inh\ L, m) \end{array}$$

Definition 7 (Common superclasses). *Define* $commSup : Cid \to Set[Cid]$*:*

$$commSup(C) \triangleq com(C.inh)$$

where $com : List[Cid] \to Set[Cid]$ *is defined by:*

$$\begin{array}{lll} com(nil) & \triangleq & \emptyset \\ com(C\ L) & \triangleq & (sup(C) \cap sup(L)) \cup com(L) \end{array}$$

and $sup : List[Cid] \to Set[Cid]$ *is defined by:*

$$\begin{array}{ll} sup(nil) & \triangleq \emptyset \\ sup(C\ L) & \triangleq \{C\} \cup sup(C.inh\ L) \end{array}$$

Definition 8 (Leftmost superclass). *Define* $lm : List[Cid] \times Cid \to Cid$ *as a partial function by :*

$$\begin{array}{lll} lm(nil, C) & \triangleq \bot & \\ lm(D\ L, C) & \triangleq D & \text{if } D \leq C \\ lm(D\ L, C) & \triangleq lm(L, C) & \text{otherwise} \end{array}$$

Definition 9 (Diamond requirement collection). *Define* $dreq : Cid \times Cid \times Mid \to Set[Prop]$ *as a partial function by:*

$$dreq(G, D, m) \triangleq ldreq(G.inh, lm(G.inh, D), D, m)$$

where $ldreq : List[Cid] \times Cid \times Cid \times Mid \to Set[Prop]$ *is defined by:*

$$\begin{array}{lll} ldreq(L, \bot, D, m) & \triangleq \bot & \\ ldreq(nil, E, D, m) & \triangleq \emptyset & \\ ldreq(C\ L, E, D, m) & \triangleq R(C, D, m) \cup ldreq(C.inh\ L, E, D, m) & \text{if } C < D \text{ and } E \not\leq C \\ ldreq(C\ L, E, D, m) & \triangleq ldreq(L, E, D, m) & \text{otherwise} \end{array}$$

Definition 10 (Entailment). *Let* p' *denote an expression* p *with all occurrences of fields* f *substituted by* f'*, avoiding name capture. Let* (p, q) *and* (r, s) *be specifications and let* $\mathcal{U}$ *and* $\mathcal{V}$ *denote the sets* $\{(p_i, q_i) \mid 1 \leq i \leq n\}$ *of specifications and* $\{(r_i, s_i) \mid 1 \leq i \leq m\}$. Entailment *is defined by*

1. $(p, q) \rightarrowtriangle (r, s) \triangleq (\forall \overline{z}_1 \,.\, p \Rightarrow q') \Rightarrow (\forall \overline{z}_2 \,.\, r \Rightarrow s')$,
   where $\overline{z}_1$ *and* $\overline{z}_2$ *are the logical variables in* (p, q) *and* (r, s)*, respectively.*
2. $\mathcal{U} \rightarrowtriangle (r, s) \triangleq (\bigwedge_{1 \leq i \leq n} (\forall \overline{z}_i \,.\, p_i \Rightarrow q'_i)) \Rightarrow (\forall z \,.\, r \Rightarrow s')$.
3. $\mathcal{U} \rightarrowtriangle \mathcal{V} \triangleq \bigwedge_{1 \leq i \leq m} \mathcal{U} \rightarrowtriangle (r_i, s_i)$.

B Auxiliary Lemmas

Lemma 2. *Let* $D, E, G : Cid$ *be classes such that* $G < D$*, and let* $E = lm(G.inh, D)$*. Then, for any* $m : Mid$ *such that* $m \notin G.mtds$*, we have* $pbind(G, D, m) = pbind(E, D, m)$.

Proof. Follows directly from Def. 2 of *pbind*, since E is the leftmost class in $G.inh$ that is below D and *pbind* binds below D.

Lemma 3. *Let* $D, E, G : Cid$ *be classes such that* $G < D$ *and* $E = lm(G.inh, D)$*. Let* $C : Cid$ *be a class such that* $G < C < E$ *and* $E \not\leq C$*. Then* $R(C, D, m) \subseteq dreq(G, D, m)$ *for any* $m : Mid$*.*

Proof. The function $dreq(G, D, m)$ will in one step evaluate to $ldreq(G.inh, E, D, m)$. This function will traverse the class hierarchy (strictly) above G, but ignore classes that are above E, or not below D. For all other classes, the function will return the union of $R(H, D, m)$ for all H in the set $\{H : Cid \mid G < H < D\} \setminus sup(E)$. It follows from the premises that C is in this set.

C Structural Inference Rules

$$\frac{\mathcal{E} \vdash [\langle C : O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{verify}(D, m, \emptyset) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (NoReq)}$$

$$\frac{\mathcal{E} \vdash [\langle C : O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeMtds}(\emptyset) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (NoMtds)}$$

$$\frac{\mathcal{E} \vdash [\langle C : O\rangle ; \mathcal{S}] \cdot \mathcal{A} \qquad t \text{ do not contain call statements}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeOutline}(D, t) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (Skip)}$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{verify}(D, m, \overline{R_1}) \cdot \mathit{verify}(D, m, \overline{R_2}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{verify}(D, m, \overline{R_1} \cup \overline{R_2}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (DecompReq)}$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{analyzeOutline}(D, t_1) \cdot \mathit{analyzeOutline}(D, t_2) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeOutline}(D, t_1; t_2) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (DecompCalls)}$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{analyzeMtds}(\overline{M_1}) \cdot \mathit{analyzeMtds}(\overline{M_2}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeMtds}(\overline{M_1} \cup \overline{M_2}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (DecompMtds)}$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{supCls}(\overline{D_1}) \cdot \mathit{supCls}(\overline{D_2}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{supCls}(\overline{D_1} \cup \overline{D_2}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (DecompSup)}$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{supMtd}(D, \overline{m_1}) \cdot \mathit{supMtd}(D, \overline{m_2}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{supMtd}(D, \overline{m_1} \cup \overline{m_2}) \cdot O\rangle ; \mathcal{S}] \cdot \mathcal{A}} \text{ (DecompSupMtd)}$$