Verifying C++ Dynamic Binding

Niels Mommen
imec-DistriNet Research Group, KU Leuven
Department of Computer Science
Leuven, Belgium
niels.mommen@kuleuven.be

Bart Jacobs
imec-DistriNet Research Group, KU Leuven
Department of Computer Science
Leuven, Belgium
bart.jacobs@kuleuven.be

ABSTRACT

We propose an approach for modular verification of programs written in an object-oriented language where, like in C++, the same virtual method call is bound to different methods at different points during the construction or destruction of an object. Our separation logic combines Parkinson and Bierman's abstract predicate families with essentially explicitly tracking each subobject's vtable pointer. Our logic supports polymorphic destruction. Virtual inheritance is not yet supported. We formalised our approach and implemented it in our VeriFast tool for semi-automated modular formal verification of C++ programs.

1 INTRODUCTION

Despite the rise of safer alternatives like Rust, C++ is still an extremely widely-used language, often for code that is safety- or security-critical [1, 3, 5]. Modular formal verification can be a powerful tool for gaining assurance that programs satisfy critical safety or security requirements; however, so far no modular formal verification approaches have been proposed for C++ programs. There has been much work on modular verification of C programs, and on modular verification of object-oriented languages, including languages with multiple inheritance. However, these are not directly applicable to C++, in large part due to its peculiar semantics of dynamic binding during object construction and destruction. In this paper, we propose what we believe to be the first Hoare logic [4] for an object-oriented language that reflects C++'s semantics of dynamic binding in the presence of constructors and destructors. Our separation logic [12] combines Parkinson and Bierman's abstract predicate families [9, 10] with essentially explicitly tracking each subobject's vtable pointer. Our logic also supports polymorphic destruction (applying the **delete** operator to an expression whose static type is a supertype of its dynamic type). Virtual inheritance, however, is not yet supported.

The remainder of this paper is structured as follows. In §2 we introduce the syntax and operational semantics of the minimal C++-like language that we will use to present our approach. In §3 we introduce our separation logic. In §4 we illustrate an example annotated with a proof outline of our program logic. We end with a discussion of related work (§5) and a conclusion (§6).

2 A MINIMAL C++-LIKE LANGUAGE

The syntax of our minimal object-oriented programming language is shown in Fig. 1. We assume infinite disjoint sets $\mathcal{C}$ of class names, $\mathcal{M}$ of method names, $\mathcal{F}$ of field names, and $\mathcal{X}$ of variable names, ranged over by symbols C, m, f, and x, respectively. We assume $\mathtt{this} \in \mathcal{X}$. For now, we also assume a set $\mathcal{A}$ of assertions, ranged over by P and Q. We will define the syntax of assertions in §3.

$$
\begin{array}{rcl}
v & ::= & \mathtt{null} \mid o \\
e & ::= & v \mid x \mid e \rightarrow f \mid \mathtt{new}\ C(\overline{e}) \mid (C*)\ e \\
c & ::= & \mathtt{let}\ x := e\ \mathtt{in}\ c \mid \mathtt{delete}\ e \mid e \rightarrow f := e \\
 & & \mid e \rightarrow C{::}m(\overline{e}) \mid e \rightarrow m(\overline{e}) \mid c; c \mid \mathtt{skip} \\
\mathit{field} & ::= & f := \mathtt{null}; \\
\mathit{pred} & ::= & \mathtt{pred}\ p(\overline{x}) = P; \\
\mathit{meth} & ::= & \mathtt{virtual}\ m(\overline{x})\ \mathtt{req}\ P\ \mathtt{ens}\ Q\ \{c\} \\
\mathit{ctor} & ::= & C(\overline{x})\ \mathtt{req}\ P\ \mathtt{ens}\ Q : \overline{C(\overline{e})}\ \{c\} \\
\mathit{dtor} & ::= & \mathtt{virtual}\ \mathtt{\sim}C()\ \mathtt{req}\ P\ \mathtt{ens}\ Q\ \{c\} \\
\mathit{class} & ::= & \mathtt{class}\ C : \overline{C}\ \{\overline{\mathit{field}}\ \overline{\mathit{pred}}\ \mathit{ctor}\ \mathit{dtor}\ \overline{\mathit{meth}}\}; \\
\mathit{prog} & ::= & \overline{\mathit{class}}\ c
\end{array}
$$

Figure 1: Syntax of the minimal language

A program consists of a sequence of class definitions, followed by a command that gets executed when the program starts. For the remainder of the formal treatment, we fix a program prog. Whenever we use a class class as a proposition, we mean $\mathit{class} \in \mathit{prog}$.

For all C, we define the set $\mathit{bases}(C)$ as the set of all direct base classes of C:

$$\mathtt{class}\ C : \overline{C}\ \{\cdots\} \Rightarrow \mathit{bases}(C) = \{\overline{C}\}$$

An *object pointer* $o \in \mathcal{O}$ is either an *allocation pointer* of the form $(\mathit{id} : C*)$ where $\mathit{id} \in \mathbb{N}$ is an *allocation identifier*, or a *subobject pointer* of the form o_C where o is an object pointer:

$$o \in \mathcal{O} ::= (\mathit{id} : C*) \mid o_C$$

We use notation $o :_{\mathsf{st}} C$ to denote the object pointed to by o has static type C:

$$o :_{\mathsf{st}} C \Leftrightarrow (\exists \mathit{id}.\ o = (\mathit{id} : C*)) \vee (\exists o'.\ o = o'_C)$$

Notice that for simplicity, the values of our language are only the object pointers and the $\mathtt{null}$ value. Furthermore, fields and other variables are untyped and hold scalar values only. That is, objects never appear on the stack or as (non-base) subobjects of other objects.

We define a heap, ranged over by h, as a finite set of resources. Resources, ranged over by α, are defined as follows:

$$\alpha ::= \mathtt{alloc}(\mathit{id}) \mid \mathtt{cted}(o) \mid o \rightarrow f \mapsto v \mid o :_{\mathsf{dyn}} C$$

where $\mathtt{alloc}(\mathit{id})$ means that an object with allocation identifier id has been allocated, $\mathtt{cted}(o)$ means that the object pointed to by o (always an allocation pointer) has been fully constructed and is not yet being destructed. Resource $o \rightarrow f \mapsto v$ means that field f of the object pointed to by o has value v, and $o :_{\mathsf{dyn}} C$ means that the dynamic type of the object pointed to by o (always a *leaf object*, whose class has no bases)[1] is C.

We define $dtype(o, C)$ as the set of all $:_{\mathsf{dyn}}$ resources of its leaf base objects, or its own $:_{\mathsf{dyn}}$ resource when it does not have any base objects, given that $o :_{\mathsf{st}} C'$:

$$dtype(o, C) \stackrel{\text{def}}{=} \begin{cases} \{o :_{\mathsf{dyn}} C\} & bases(C') = \emptyset \\ \bigcup\limits_{1 \le i \le n} dtype(o_{C_i}, C) & bases(C') = C_1 \ldots C_n \end{cases}$$

We say an object pointed to by o has dynamic type C in a heap h if and only if $dtype(o, C) \subseteq h$. Notice that a non-leaf object has dynamic type C if and only if all of its bases have dynamic type C. As we will see, dynamically dispatched calls on an object o are dispatched to the dynamic type of o. If an object o has no dynamic type in our language, dynamically dispatched calls get stuck. As we will also see, an object o has no dynamic type while its bases are being constructed or destructed, nor while unrelated (i.e. neither enclosed nor enclosing) subobjects of the allocation are being constructed or destructed. It has a dynamic type only while its own constructor's or destructor's body, or the body of an enclosing object's constructor or destructor is executing, and between the point where its enclosing allocation is fully constructed and the point where it starts being destructed.

We use $o \downarrow C$ (*o downcast to C*) to denote the pointer to the enclosing object of class C of the object pointed to by o:

$$\frac{o :_{\mathsf{st}} C}{o \downarrow C = o} \qquad \frac{o \downarrow C = o'}{o_{C'} \downarrow C = o'}$$

We use $h, e \Downarrow h', v$ to denote that when evaluated in heap h, expression e evaluates to value v and post-heap h'. Similarly, we use $h, c \Downarrow h'$ and $h, o \rightarrow C(\overline{e}) \Downarrow h'$ and $h, o \rightarrow \mathord{\sim}C() \Downarrow h'$ to denote that command c, constructor call $o \rightarrow C(\overline{e})$, and destructor call $o \rightarrow \mathord{\sim}C()$, when executed in heap h, terminate with post-heap h', respectively. These judgments are defined by mutual induction; we show selected rules in Fig. 2. (The complete set of rules can be found in the appendix.)

Notice, first of all, that a statically dispatched call $e \rightarrow C{::}m(\overline{e})$ gets stuck if class C does not declare a method m, even if some base does declare such a method: in our minimal language, classes do not inherit methods from their bases. The same holds for dynamically dispatched calls.[2]

Evaluation of $\mathtt{new}\ C(\overline{e})$ picks an unused allocation identifier id and produces (i.e. adds to the heap) $\mathtt{alloc}(id)$ to mark it as used, then executes the constructor call, and finally produces $\mathtt{cted}(o)$ to mark o as fully constructed.

Executing a constructor call $o \rightarrow C(\overline{e})$ is somewhat involved. If C has no bases, the argument expressions are evaluated, the fields are produced, $o :_{\mathsf{dyn}} C$ is produced, and the constructor body is executed. Considered together with ODynamicDispatch, this means that dynamically dispatched calls on `this` in the constructor body are dispatched to class C itself, even if C is not the most derived class of the allocation.

Now consider the case where C does have bases. Executing constructor call $o \rightarrow C(\overline{e})$ evaluates the argument expressions and then executes each base class' constructor on the corresponding base subobject. After executing the constructor for base C_i, $dtype(o_{C_i}, C_i)$ is consumed (i.e. removed from the heap); after all base subobjects have been initialized, $dtype(o, C)$ is produced. This means that, during execution of the body of the constructor of class C, dynamically dispatched calls on o or on any base subobject of o are dispatched to class C. After an allocation of class C is fully constructed, and until it starts being destructed, its dynamic type (and that of all of its subobjects) is C.

Execution of a destructor call $o \rightarrow \mathord{\sim}C()$ performs the exact reverse process: it executes the destructor body, consumes $dtype(o, C)$ and the fields, and destructs the base subobjects. Before destructing the subobject for base C_i, $dtype(O_{C_i}, C_i)$ is produced, so that during execution of the body of the destructor of an object o of class C, dynamically dispatched calls on o are dispatched to class C. After destruction of an allocation completes, only the `alloc` resource remains, to ensure that no future allocation is assigned the same identifier.[3]

Deleting an object gets stuck unless its enclosing allocation is fully constructed and is not yet being destructed, as indicated by the presence of the `cted` resource. Since this resource always holds an allocation pointer, it is always the entire allocation that is destroyed, even if the argument to `delete` is a pointer to a subobject.

We use judgments $h, e\ \mathtt{div}$ and $h, c\ \mathtt{div}$ and $h, o \rightarrow C(\overline{e})\ \mathtt{div}$ and $h, o \rightarrow \mathord{\sim}C()\ \mathtt{div}$ to denote that an expression, command, constructor call, or destructor call diverges (i.e. runs forever without terminating or getting stuck), respectively. These judgments' definitions can be derived mechanically [2] from the definitions of the termination judgments and are therefore elided.

3 A PROGRAM LOGIC FOR C++ DYNAMIC BINDING

A class definition in our language includes a list of abstract predicates. A predicate declaration in a class defines its entry for the corresponding predicate family, i.e., a class defines its own definition for the abstract predicate, which can be overridden by derived classes. As we will see, predicate assertions involve a *class index* to refer to the definition of the predicate declared in that class.

We use a context Γ, which is a sequence of class definitions.

3.1 Assertions

Predicate definitions, method specifications, constructor specifications, and destructor specifications consist of assertions, ranged over by P and Q:

$$\begin{array}{rcl} P, Q & ::= & \mathtt{true} \mid \mathtt{false} \mid P \land Q \mid P \lor Q \mid P * Q \mid \exists x.\ P \\ & & \mid \varepsilon \rightarrow f \mapsto \varepsilon \mid \varepsilon \rightarrow p_\varepsilon(\overline{\varepsilon}) \mid \mathtt{cted}(\varepsilon, \varepsilon) \mid \varepsilon :_{\mathsf{dyn}} \varepsilon \\ \nu & ::= & v \mid C \\ \varepsilon & ::= & x \mid \nu \end{array}$$

[1]This corresponds to the fact that in C++, objects that have polymorphic base subobjects can reuse the (first) polymorphic base subobject's vtable pointer. Note: in this paper, for simplicity we do not consider non-polymorphic classes, i.e. classes that do not declare or inherit any virtual members.

[2]Of course, a program that does rely on method inheritance can be trivially translated into our minimal language by inserting overrides that simply delegate to the appropriate base. Importantly, however, those overrides will have to be verified as part of the correctness proof (see §3); their correctness does not hold automatically.

[3]This reflects the fact that pointers in C++ become invalid permanently after the allocation they point to is deallocated, even if some future allocation happens to reuse the same address.

OUpcast

$$\frac{h, e \Downarrow h', o \qquad o :_{\mathsf{st}} C \qquad C' \in bases(C)}{h, (C'*)\ e \Downarrow h', o_{C'}}$$

ONew

$$\frac{\begin{array}{c} o = (id : C*) \qquad id = \min\{id \mid \mathtt{alloc}(id) \notin h\} \\ h \uplus \{\!\!\{\mathtt{alloc}(id)\}\!\!\}, o \rightarrow C(\overline{e}) \Downarrow h' \end{array}}{h, \mathtt{new}\ C(\overline{e}) \Downarrow h' \uplus \{\!\!\{\mathtt{cted}(o)\}\!\!\}, o}$$

ODelete

$$\frac{\begin{array}{c} o' = o \downarrow C \qquad h, e \Downarrow h' \uplus \{\!\!\{\mathtt{cted}(o')\}\!\!\}, o \\ h', o' \rightarrow \mathord{\sim} C() \Downarrow h'' \end{array}}{h, \mathtt{delete}(e) \Downarrow h''}$$

OStaticDispatch

$$\frac{\begin{array}{c} \mathtt{class}\ C \cdots \{ \cdots \mathtt{virtual}\ m(\overline{x})\{c\} \cdots \} \qquad o :_{\mathsf{st}} C \\ h, e \Downarrow h', o \qquad h', \overline{e} \Downarrow h'', \overline{v} \qquad h'', c[o/\mathtt{this}, \overline{v}/\overline{x}] \Downarrow h''' \end{array}}{h, e \rightarrow C{::}m(\overline{e}) \Downarrow h'''}$$

ODynamicDispatch

$$\frac{\begin{array}{c} \mathtt{class}\ C \cdots \{ \cdots \mathtt{virtual}\ m(\overline{x})\{c\} \cdots \} \qquad h, e \Downarrow h', o \qquad o' = o \downarrow C \\ h', \overline{e} \Downarrow h'', \overline{v} \qquad dtype(o, C) \subseteq h'' \qquad h'', c[o'/\mathtt{this}, \overline{v}/\overline{x}] \Downarrow h''' \end{array}}{h, e \rightarrow m(\overline{e}) \Downarrow h'''}$$

OConstruct

$$\frac{\begin{array}{c} \mathtt{class}\ C : C_1 \ldots C_n \{ \overline{f \coloneqq \mathtt{null};} \cdots C(\overline{x}) : C_1(\overline{e_1}) \ldots C_n(\overline{e_n})\{c\} \cdots \} \\ h, \overline{e} \Downarrow h_0, \overline{v} \\ h_0, o_{C_1} \rightarrow C_1(\overline{e_1}[o/\mathtt{this}, \overline{v}/\overline{x}]) \Downarrow h_1 \uplus dtype(o_{C_1}, C_1) \\ \ldots \\ h_{n-1}, o_{C_n} \rightarrow C_n(\overline{e_n}[o/\mathtt{this}, \overline{v}/\overline{x}]) \Downarrow h_n \uplus dtype(o_{C_n}, C_n) \\ h_n \uplus \{\!\!\{\overline{o \rightarrow f \mapsto \mathtt{null}}\}\!\!\} \uplus dtype(o, C), c[o/\mathtt{this}, \overline{v}/\overline{x}] \Downarrow h' \end{array}}{h, o \rightarrow C(\overline{e}) \Downarrow h'}$$

ODestruct

$$\frac{\begin{array}{c} \mathtt{class}\ C : C_1 \ldots C_n \{ \overline{f \coloneqq \mathtt{null};} \cdots \mathtt{virtual}\ \mathord{\sim}C()\{c\} \cdots \} \\ h, c[o/\mathtt{this}] \Downarrow h_n \uplus dtype(o, C) \uplus \{\!\!\{\overline{o \rightarrow f \mapsto v}\}\!\!\} \\ h_n \uplus dtype(o_{C_n}, C_n), o_{C_n} \rightarrow \mathord{\sim}C_n() \Downarrow h_{n-1} \\ \ldots \\ h_1 \uplus dtype(o_{C_1}, C_1), o_{C_1} \rightarrow \mathord{\sim}C_1() \Downarrow h_0 \end{array}}{h, o \rightarrow \mathord{\sim}C() \Downarrow h_0}$$

Figure 2: Operational semantics of the minimal language related to allocation and deallocation, construction and destruction, and method dispatching.

where $P * Q$ is the separating conjunction of assertions P and Q, which informally means that assertion P and Q must be satisfied in disjoint portions of the heap. Assertion $\varepsilon \rightarrow p_{\varepsilon'}(\overline{\varepsilon''})$ is a predicate assertion p with class index ε' on the target object pointed to by ε.

We show the semantics of the most interesting assertions:

$$\begin{array}{lcl} I, h \vDash o \rightarrow p_C(\overline{v}) & \Leftrightarrow & \exists o'.\ o \downarrow C = o' \wedge (h, o', p, C, \overline{v}) \in I \\ I, h \vDash \mathtt{cted}(o, C) & \Leftrightarrow & \exists o'.\ o \downarrow C = o' \wedge \mathtt{cted}(o') \in h \\ I, h \vDash o :_{\mathsf{dyn}} C & \Leftrightarrow & dtype(o, C) \subseteq h \\ I, h \vDash o \rightarrow f \mapsto v & \Leftrightarrow & o \rightarrow f \mapsto v \in h \end{array}$$

where $I, h \vDash P$ means that assertion P is satisfied, given heap h and interpretation of predicates I. An interpretation of predicates is the least fixpoint of the program's predicate definitions considered together.

We define the assertion weakening relation $\Gamma \vdash P \Rightarrow_a Q$ by induction, where every judgment $P \Rightarrow_a Q$ should be read as $\Gamma \vdash P \Rightarrow_a Q$:

ADyntype

$$\frac{o :_{\mathsf{st}} C \qquad bases(C) = C_1 \ldots C_n \qquad n > 0}{o :_{\mathsf{dyn}} C' \Leftrightarrow_a o_{C_1} :_{\mathsf{dyn}} C' * \ldots * o_{C_n} :_{\mathsf{dyn}} C'}$$

AFrame

$$\frac{P \Rightarrow_a P'}{P * Q \Rightarrow_a P' * Q}$$

ATrans

$$\frac{P \Rightarrow_a P' \qquad P' \Rightarrow_a P''}{P \Rightarrow_a P''}$$

AMoveCted

$$\frac{\begin{array}{c} o :_{\mathsf{st}} C \\ C' \in bases(C) \qquad C' \neq C'' \end{array}}{\mathtt{cted}(o, C'') \Leftrightarrow_a \mathtt{cted}(o_{C'}, C'')}$$

AImply

$$\frac{\forall I, h.\ I, h \vDash P \Rightarrow I, h \vDash P'}{P \Rightarrow_a P'}$$

AMovePred

$$\frac{\begin{array}{c} o :_{\mathsf{st}} C \\ C' \in bases(C) \qquad C' \neq C'' \end{array}}{o \rightarrow p_{C''}(\overline{v}) \Leftrightarrow_a o_{C'} \rightarrow p_{C''}(\overline{v})}$$

APredDef

$$\frac{o :_{\mathsf{st}} C \qquad \mathtt{class}\ C \cdots \{ \cdots \mathtt{pred}\ p(\overline{x}) = P \cdots \} \in \Gamma}{o \rightarrow p_C(\overline{v}) \Leftrightarrow_a P[o/\mathtt{this}, \overline{v}/\overline{x}]}$$

Weakening rule APredDef allows to switch between a predicate assertion and the definition of the predicate corresponding to the class index. The class index must be a class name declared in the program.

AMovePred and AMoveCted allow to *transfer* predicate and `cted` assertions between base and derived objects. It is not possible to transfer such an assertion to an object whose dynamic type is a subtype of the predicate index and allocation class, respectively.

Weakening rule ADyntype states that the dynamic type assertion of a non-leaf object can be exchanged for all dynamic type assertions of its direct base objects. This means that the dynamic type of a base object can be retrieved if the dynamic type of its direct derived object is known. The other way around, it is possible to derive the dynamic type of a derived object if the dynamic type of all its direct base classes is known.

3.2 Expression and command verification

The verification rules for the most interesting expressions and commands are listed in Fig. 3, together with the verification rules for constructor and destructor invocations. These rules are related to object allocation and deallocation, and static and dynamic dispatching. (The complete set of verification rules can be found in the appendix).

In method and destructor specifications, we use special variable θ to refer to the class of the target object of the call. This variable is assumed to be equal to the containing class during verification of the method or destructor. This is sound, because we require that a class overrides all methods of all its direct base classes, as we will later see. Hence when a call is dynamically dispatched, it will always be bound to the method declared in the class corresponding with the dynamic type of the target object.

Variable θ is substituted with the dynamic type of the target object and the static type of the target object during verification of dynamically dispatched calls and statically dispatched calls, respectively. This mechanism allows to use the specification for the method or destructor in the class corresponding to the static type of the method or destructor target.

3.3 Constructor verification

The verification rule for constructors follows OConstruct from our operational semantics: the direct base constructor invocations are verified in order of inheritance, prior to initializing the fields of the object and verifying the command in the constructor's body. Virtual calls are always dispatched to the (sub)object under construction.

$$\frac{\begin{array}{c}\forall o :_{\mathtt{st}} C, \overline{v}.\\ P[\overline{v}/\overline{x}] = P_0\\ \{P_0\}\ o_{C_1} \rightarrow C_1(\overline{e_1}[o/\mathtt{this}, \overline{v}/\overline{x}])\ \{P_1 * o_{C_1} :_{\mathtt{dyn}} C_1\}\\ \dots\\ \{P_{n-1}\}\ o_{C_n} \rightarrow C_n(\overline{e_n}[o/\mathtt{this}, \overline{v}/\overline{x}])\ \{P_n * o_{C_n} :_{\mathtt{dyn}} C_n\}\\ \{P_n * \overline{o \rightarrow f \mapsto \mathtt{null}} * o :_{\mathtt{dyn}} C\}\ c[o/\mathtt{this}, \overline{v}/\overline{x}]\ \{Q[o/\mathtt{this}, \overline{v}/\overline{x}]\}\end{array}}{\Gamma \vdash C(\overline{x})\ \mathtt{req}\ P\ \mathtt{ens}\ Q : C_1(\overline{e_1}) \dots C_n(\overline{e_n})\ \{c\}\ \text{correct in } C}$$

3.4 Behavioral subtyping

We follow Parkinson and Bierman's approach [10] to check whether specifications of overriding methods satisfy behavioral subtyping. A specification $\{P_D\}\_\{Q_D\}$ of an overriding method in derived class D *implies* a specification $\{P_B\}\_\{Q_B\}$ of a method in base class B, if for all commands c, values $\overline{v}$ and object pointers $o :_{\mathtt{st}} B$ with a well-defined downcast $o' = o \downarrow D$ that satisfy $\{P_D[S_D]\}\ c\ \{Q_D[S_D]\}$, it holds that $\{P_B[S_B]\}\ c\ \{Q_B[S_B]\}$ is also satisfied, with $S_B = o/\mathtt{this}, D/\theta, \overline{v}/\overline{x}$ and $S_D = o'/\mathtt{this}, D/\theta, \overline{v}/\overline{x}$. This holds when a proof tree exists using the structural rules of Hoare and Separation logic, with leaves $\Gamma \vdash \{P_D[S_D]\}\_\{Q_D[S_D]\}$ and root $\Gamma \vdash \{P_B[S_B]\}\_\{Q_B[S_B]\}$:

$$\frac{\dfrac{\Gamma \vdash \{P_D[S_D]\}\_\{Q_D[S_D]\}}{\vdots}}{\Gamma \vdash \{P_B[S_B]\}\_\{Q_B[S_B]\}}$$

We use notation $\Gamma \vdash \{P_D\}\_\{Q_D\}\ {}^{D}\!\!\Rightarrow^{B}\ \{P_B\}\_\{Q_B\}$ to denote that such a proof exists.

3.5 Method verification

The verification rule for correctly overriding a method checks that (1) the specification for method m in derived class C satisfies behavioral subtyping for base class C' which also declares m, and (2) recursively checks this condition for all direct base classes of C'. We use $methods(C)$ to denote all methods declared in class C.

$$\frac{\begin{array}{c}\mathtt{class}\ C \cdots \{\cdots \mathtt{virtual}\ m(\overline{x})\ \mathtt{req}\ P\ \mathtt{ens}\ Q \cdots\} \in \Gamma\\ \mathtt{class}\ C' \cdots \{\cdots \mathtt{virtual}\ m(\overline{x})\ \mathtt{req}\ P'\ \mathtt{ens}\ Q' \cdots\} \in \Gamma\\ \Gamma \vdash \{P\}\_\{Q\}\ {}^{C}\!\!\Rightarrow^{C'}\ \{P'\}\_\{Q'\}\\ \forall C'' \in bases(C').\, m \in methods(C'') \Rightarrow\\ \Gamma \vdash \text{override of } m \text{ in } C'' \text{ correct in C}\end{array}}{\Gamma \vdash \text{override of } m \text{ in } C' \text{ correct in } C}$$

Method m in class C is correct if (1) the override check for all base classes of C that declare m succeeds and (2) the method body satisfies its specification given that the target class type is C.

$$\frac{\begin{array}{c}\forall C' \in bases(C).\, m \in methods(C') \Rightarrow\\ \Gamma \vdash \text{override of } m \text{ in } C' \text{ correct in C}\\ \forall o :_{\mathtt{st}} C, \overline{v}.\\ \{P[o/\mathtt{this}, C/\theta, \overline{v}/\overline{x}]\}\ c[o/\mathtt{this}, \overline{v}/\overline{x}]\ \{Q[o/\mathtt{this}, C/\theta, \overline{v}/\overline{x}]\}\end{array}}{\Gamma \vdash m(\overline{x})\ \mathtt{req}\ P\ \mathtt{ens}\ Q\ \{c\}\ \text{correct in } C}$$

3.6 Destructor verification

The verification rule for correctly overriding a destructor is similar to the verification rule for correctly overriding a method. The difference is that it recursively checks the rule for *all* bases because every class must declare a destructor in our language.

The verification rule for destructors again resembles the operational semantics and follows the reverse process of its corresponding constructor. The command of the body is first verified, followed by the removal of the object's fields and verification of the direct base destructor invocations in reverse order of inheritance. Virtual member invocations are dispatched to the (sub)object under destruction.

$$\frac{\begin{array}{c}\forall C' \in bases(C).\, \Gamma \vdash \text{override of destructor in } C' \text{ correct in } C\\ \forall o :_{\mathtt{st}} C.\\ bases(C) = C_1 \dots C_n \qquad P_0 = Q\\ \{P[o/\mathtt{this}, C/\theta]\}\ c[o/\mathtt{this}]\ \{P_n * \overline{o \rightarrow f \mapsto \_} * o :_{\mathtt{dyn}} C\}\\ \{P_n * o_{C_n} :_{\mathtt{dyn}} C_n\}\ o_{C_n} \rightarrow \mathord{\sim}C_n()\ \{P_{n-1}\}\\ \dots\\ \{P_1 * o_{C_1} :_{\mathtt{dyn}} C_1\}\ o_{C_1} \rightarrow \mathord{\sim}C_1()\ \{P_0\}\end{array}}{\Gamma \vdash \mathord{\sim}C()\ \mathtt{req}\ P\ \mathtt{ens}\ Q\ \{c\}\ \text{correct in } C}$$

3.7 Program verification

Verification of a class succeeds if verification for its constructor, destructor, and methods succeeds. We additionally require that a derived class overrides all methods declared in its base classes. This requirement renders our assumption sound that the dynamic type of the target object during verification of a destructor or method is the class type of the enclosing class it is declared in.

A program is correct if verification of all its classes succeeds, and its main command is verifiable given an empty heap.

$$\frac{prog = \overline{class}\ c \qquad \vdash \overline{class}\ \text{correct} \qquad \vdash \{\mathtt{true}\}\ c\ \{\mathtt{true}\}}{\vdash \text{program correct}}$$

Theorem 1 (Soundness). *Given that the program is correct, the main command, when executed in the empty heap, does not get stuck (i.e. it either terminates or diverges):*

$$\vdash \text{program correct} \land prog = \overline{class}\ c \Rightarrow \emptyset, c \Downarrow \_ \lor \emptyset, c\ \mathtt{div}$$

4 EXAMPLE PROOF OUTLINE

This section shows an example in our formal language, annotated with its proof outline. It illustrates a *node* class N which inherits from both a *target* class T and *source* class S. A target and source can have a source and target, respectively. A node is initially its own target and source.

HStaticDispatch
$$\frac{\mathtt{class}\ C \cdots \{\cdots \mathtt{virtual}\ m(\overline{x})\ \mathtt{req}\ P\ \mathtt{ens}\ Q \cdots\} \in \Gamma \qquad o :_{\mathsf{st}} C}{\{P[o/\mathtt{this}, C/\theta, \overline{v}/\overline{x}]\}\ o \to C{::}m(\overline{v})\ \{Q[o/\mathtt{this}, C/\theta, \overline{v}/\overline{x}]\}}$$

HNew
$$\frac{\mathtt{class}\ C \cdots \{\cdots C(\overline{x})\ \mathtt{req}\ P\ \mathtt{ens}\ Q \cdots\} \in \Gamma}{\{P[\overline{v}/\overline{x}]\}\ \mathsf{new}\ C(\overline{v})\ \{Q[\overline{v}/\overline{x}, \mathtt{result}/\mathtt{this}] * \mathtt{cted}(\mathtt{result}, C)\}}$$

HDynamicDispatch
$$\frac{\mathtt{class}\ C \cdots \{\cdots \mathtt{virtual}\ m(\overline{x})\ \mathtt{req}\ P\ \mathtt{ens}\ Q \cdots\} \in \Gamma \qquad o :_{\mathsf{st}} C}{\{o :_{\mathsf{dyn}} C' \wedge P[o/\mathtt{this}, C'/\theta, \overline{v}/\overline{x}]\}\ o \to m(\overline{v})\ \{Q[o/\mathtt{this}, C'/\theta, \overline{v}/\overline{x}]\}}$$

HDestruct
$$\frac{\mathtt{class}\ C \cdots \{\cdots \mathtt{virtual}\ \ \mathtt{\sim} C()\ \mathtt{req}\ P\ \mathtt{ens}\ Q \cdots\} \in \Gamma}{\{P[o/\mathtt{this}, C/\theta]\}\ o \to \mathtt{\sim} C()\ \{Q\}}$$

HDelete
$$\frac{o :_{\mathsf{st}} C \qquad \mathtt{class}\ C \cdots \{\cdots \mathtt{virtual}\ \ \mathtt{\sim} C()\ \mathtt{req}\ P\ \mathtt{ens}\ Q \cdots\} \in \Gamma}{\{\mathtt{cted}(o, C') * P[o/\mathtt{this}, C'/\theta]\}\ \mathtt{delete}(o)\ \{Q\}}$$

HUpcast
$$\frac{o :_{\mathsf{st}} C' \qquad C \in \mathit{bases}(C')}{\{P[o_C/\mathtt{result}]\}\ (C*)\ o\ \{P\}}$$

HConstruct
$$\frac{\mathtt{class}\ C \cdots \{\cdots C(\overline{x})\ \mathtt{req}\ P\ \mathtt{ens}\ Q \cdots\} \in \Gamma}{\{P[\overline{v}/\overline{x}])\}\ o \to C(\overline{v})\ \{Q[\overline{v}/\overline{x}, o/\mathtt{this}]\}}$$

Figure 3: Verification rules related to allocation and deallocation, construction and destruction, and method dispatching. Read judgment $\{P\}\ c\ \{Q\}$ as $\Gamma \vdash \{P\}\ c\ \{Q\}$.

The example illustrates dynamic dispatch during construction and shows that our program logic is applicable in the presence of multiple inheritance. The main command shows how our proof system can handle polymorphic deletion of objects. The proof outline for `T` is symmetric to the one shown in `S`, and is therefore omitted. Empty bodies implicitly contain a `skip` command.

class S {
t := **null**;
pred $\mathit{Sok}() = \exists t.\ \mathit{this} \to t \mapsto t$; **pred** $\mathit{sdyn}(dt) = \mathit{this} :_{\mathsf{dyn}} dt$;
S() **req** $\mathtt{true}$ **ens** $\mathit{this} \to \mathit{sdyn}_{\mathsf{S}}(\mathsf{S}) * \mathit{this} \to \mathit{Sok}_{\mathsf{S}}()$ {
$\{\mathtt{true} * \mathit{this} \to t \mapsto \mathtt{null} * \mathit{this} :_{\mathsf{dyn}} \mathsf{S}\}$
$\{\mathit{this} \to \mathit{sdyn}_{\mathsf{S}}(\mathsf{S}) * \mathit{this} \to \mathit{Sok}_{\mathsf{S}}()\}$
}
virtual ~S() **req** $\mathit{this} \to \mathit{sdyn}_{\theta}(\theta) * \mathit{this} \to \mathit{Sok}_{\theta}()$ **ens** $\mathtt{true}$ {
$\{\mathit{this} :_{\mathsf{dyn}} \mathsf{S} * \exists t.\ \mathit{this} \to t \mapsto t\}$
}
virtual setTarget(t) **req** $\mathit{this} \to \mathit{Sok}_{\theta}()$ **ens** $\mathit{this} \to \mathit{Sok}_{\theta}()$ {
$\{\exists lt.\ \mathit{this} \to t \mapsto lt\}$
$\{\mathit{this} \to t \mapsto lt\}$ **this**→t := t $\{\mathit{this} \to t \mapsto t\}$
$\{\exists lt.\ \mathit{this} \to t \mapsto lt\}$
}
};
class T {
s := **null**;
pred $\mathit{Tok}() = \exists s.\ \mathit{this} \to s \mapsto s$; **pred** $\mathit{tdyn}(dt) = \mathit{this} :_{\mathsf{dyn}} dt$;
T() {}
virtual ~T() {}
virtual setSource(s) { **this**→s := s }
};
class N : S, T {
pred $\mathit{Sok}() = \mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}() * \mathit{this}_{\mathsf{T}} \to \mathit{Tok}_{\mathsf{T}}()$;
pred $\mathit{sdyn}(dt) = \mathit{this}_{\mathsf{S}} \to \mathit{sdyn}_{\mathsf{S}}(dt) * \mathit{this}_{\mathsf{T}} \to \mathit{tdyn}_{\mathsf{T}}(dt)$;
pred $\mathit{Tok}() = \mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}() * \mathit{this}_{\mathsf{T}} \to \mathit{Tok}_{\mathsf{T}}()$;
pred $\mathit{tdyn}(dt) = \mathit{this}_{\mathsf{S}} \to \mathit{sdyn}_{\mathsf{S}}(dt) * \mathit{this}_{\mathsf{T}} \to \mathit{tdyn}_{\mathsf{T}}(dt)$;
N() **req** $\mathtt{true}$ **ens** $\mathit{this} \to \mathit{sdyn}_{\mathsf{N}}(\mathsf{N}) * \mathit{this} \to \mathit{Sok}_{\mathsf{N}}()$:
$\{\mathtt{true}\}$
$\{\mathtt{true}\}$ S() $\{\mathit{this}_{\mathsf{S}} \to \mathit{sdyn}_{\mathsf{S}}(\mathsf{S}) * \mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}()\}$
$\{\mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}() * \mathit{this}_{\mathsf{S}} :_{\mathsf{dyn}} \mathsf{S}\}$
,
$\{\mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}()\}$
$\{\mathtt{true}\}$ T() $\{\mathit{this}_{\mathsf{T}} \to \mathit{tdyn}_{\mathsf{T}}(\mathsf{T}) * \mathit{this}_{\mathsf{T}} \to \mathit{Tok}_{\mathsf{T}}()\}$
$\{\mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}() * \mathit{this}_{\mathsf{T}} \to \mathit{Tok}_{\mathsf{T}}() * \mathit{this}_{\mathsf{T}} :_{\mathsf{dyn}} \mathsf{T}\}$
{
$\{\mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}() * \mathit{this}_{\mathsf{T}} \to \mathit{Tok}_{\mathsf{T}}() * \mathit{this} :_{\mathsf{dyn}} \mathsf{N}\}$
$\{\mathit{this} :_{\mathsf{dyn}} \mathsf{N} * \mathit{this} \to \mathit{Sok}_{\mathsf{N}}()\}$
$\{\exists C.\ \mathit{this} :_{\mathsf{dyn}} C * \mathit{this} \to \mathit{Sok}_{C}()\}$
this→setTarget((T *) **this**);
$\{\mathit{this} :_{\mathsf{dyn}} C * \mathit{this} \to \mathit{Sok}_{C}()\}$
$\{\mathit{this} :_{\mathsf{dyn}} \mathsf{N} * \mathit{this} \to \mathit{Sok}_{\mathsf{N}}()\}$
$\{\mathit{this} :_{\mathsf{dyn}} \mathsf{N} * \mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}() * \mathit{this}_{\mathsf{T}} \to \mathit{Tok}_{\mathsf{T}}()\}$
$\{\mathit{this} :_{\mathsf{dyn}} \mathsf{N} * \mathit{this} \to \mathit{Tok}_{\mathsf{N}}()\}$
$\{\exists C.\ \mathit{this} :_{\mathsf{dyn}} C * \mathit{this} \to \mathit{Tok}_{C}()\}$
this→setSource((S *) **this**)
$\{\mathit{this} :_{\mathsf{dyn}} C * \mathit{this} \to \mathit{Tok}_{C}()\}$
$\{\mathit{this} :_{\mathsf{dyn}} \mathsf{N} * \mathit{this} \to \mathit{Tok}_{\mathsf{N}}()\}$
$\{\mathit{this} :_{\mathsf{dyn}} \mathsf{N} * \mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}() * \mathit{this}_{\mathsf{T}} \to \mathit{Tok}_{\mathsf{T}}()\}$
$\{\mathit{this} :_{\mathsf{dyn}} \mathsf{N} * \mathit{this} \to \mathit{Sok}_{\mathsf{N}}()\}$
$\{\mathit{this}_{\mathsf{S}} :_{\mathsf{dyn}} \mathsf{N} * \mathit{this}_{\mathsf{T}} :_{\mathsf{dyn}} \mathsf{N} * \mathit{this} \to \mathit{Sok}_{\mathsf{N}}()\}$
$\{\mathit{this}_{\mathsf{S}} \to \mathit{sdyn}_{\mathsf{S}}(\mathsf{S}) * \mathit{this}_{\mathsf{T}} \to \mathit{tdyn}_{\mathsf{T}}(\mathsf{T}) * \mathit{this} \to \mathit{Sok}_{\mathsf{N}}()\}$
$\{\mathit{this} \to \mathit{sdyn}_{\mathsf{N}}(\mathsf{N}) * \mathit{this} \to \mathit{Sok}_{\mathsf{N}}()\}$
}
virtual ~N() **req** $\mathit{this} \to \mathit{sdyn}_{\theta}(\theta) * \mathit{this} \to \mathit{Sok}_{\theta}()$ **ens** $\mathtt{true}$ {
$\{\mathit{this}_{\mathsf{S}} \to \mathit{sdyn}_{\mathsf{S}}(\mathsf{N}) * \mathit{this}_{\mathsf{T}} \to \mathit{tdyn}_{\mathsf{T}}(\mathsf{N}) * \mathit{this} \to \mathit{Sok}_{\mathsf{N}}()\}$
$\{\mathit{this}_{\mathsf{S}} :_{\mathsf{dyn}} \mathsf{N} * \mathit{this}_{\mathsf{T}} :_{\mathsf{dyn}} \mathsf{N} * \mathit{this} \to \mathit{Sok}_{\mathsf{N}}()\}$
$\{\mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}() * \mathit{this}_{\mathsf{T}} \to \mathit{Tok}_{\mathsf{T}}() * \mathit{this} :_{\mathsf{dyn}} \mathsf{N}\}$
}
$\{\mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}() * \mathit{this}_{\mathsf{T}} \to \mathit{Tok}_{\mathsf{T}}() * \mathit{this}_{\mathsf{T}} :_{\mathsf{dyn}} \mathsf{T}\}$
$\{\mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}() * \mathit{this}_{\mathsf{T}} \to \mathit{Tok}_{\mathsf{T}}() * \mathit{this}_{\mathsf{T}} \to \mathit{tdyn}_{\mathsf{T}}(\mathsf{T})\}$
$\{\mathit{this}_{\mathsf{T}} \to \mathit{tdyn}_{\mathsf{T}}(\mathsf{T}) * \mathit{this}_{\mathsf{T}} \to \mathit{Tok}_{\mathsf{T}}()\}$ $\mathit{this}_{\mathsf{T}} \to \mathtt{\sim}\mathsf{T}()$ $\{\mathtt{true}\}$
$\{\mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}()\}$
,
$\{\mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}() * \mathit{this}_{\mathsf{S}} :_{\mathsf{dyn}} \mathsf{S}\}$
$\{\mathit{this}_{\mathsf{S}} \to \mathit{sdyn}_{\mathsf{S}}(\mathsf{S}) * \mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}()\}$
$\{\mathit{this}_{\mathsf{S}} \to \mathit{sdyn}_{\mathsf{S}}(\mathsf{S}) * \mathit{this}_{\mathsf{S}} \to \mathit{Sok}_{\mathsf{S}}()\}$ $\mathit{this}_{\mathsf{S}} \to \mathtt{\sim}\mathsf{S}()$ $\{\mathtt{true}\}$
$\{\mathtt{true}\}$
virtual setTarget(t) **req** $\mathit{this} \to \mathit{Sok}_{\theta}()$ **ens** $\mathit{this} \to \mathit{Sok}_{\theta}()$ {

$\{this_{\mathsf{S}} \rightarrow Sok_{\mathsf{S}}() * this_{\mathsf{T}} \rightarrow Tok_{\mathsf{T}}()\}$
$\{this_{\mathsf{S}} \rightarrow Sok_{\mathsf{S}}()\}$ **this**→S::setTarget(t) $\{this_{\mathsf{S}} \rightarrow Sok_{\mathsf{S}}()\}$
$\{this_{\mathsf{S}} \rightarrow Sok_{\mathsf{S}}() * this_{\mathsf{T}} \rightarrow Tok_{\mathsf{T}}()\}$
}
virtual setSource(s) **req** $this \rightarrow Tok_{\theta}()$ **ens** $this \rightarrow Tok_{\theta}()$ {
$\{this_{\mathsf{S}} \rightarrow Sok_{\mathsf{S}}() * this_{\mathsf{T}} \rightarrow Tok_{\mathsf{T}}()\}$
$\{this_{\mathsf{T}} \rightarrow Tok_{\mathsf{T}}()\}$ **this**→T::setSource(s) $\{this_{\mathsf{T}} \rightarrow Tok_{\mathsf{T}}()\}$
$\{this_{\mathsf{S}} \rightarrow Sok_{\mathsf{S}}() * this_{\mathsf{T}} \rightarrow Tok_{\mathsf{T}}()\}$
}
};
$\{\mathtt{true}\}$
$\{\mathtt{true}\}$
let n := **new** N() **in**
$\{n \rightarrow sdyn_{\mathsf{N}}(\mathsf{N}) * n \rightarrow Sok_{\mathsf{N}}() * \mathtt{cted}(n, \mathsf{N})\}$
let s := (S∗) n **in**
$\{s \rightarrow sdyn_{\mathsf{N}}(\mathsf{N}) * s \rightarrow Sok_{\mathsf{N}}() * \mathtt{cted}(s, \mathsf{N})\}$
$\{\exists C.\ \mathtt{cted}(s, C) * s \rightarrow sdyn_{C}(C) * s \rightarrow Sok_{\mathsf{C}}()\}$
delete s
$\{\mathtt{true}\}$
$\{\mathtt{true}\}$
$\{\mathtt{true}\}$

The proof that the specification of `~N` implies the specification of `~T`, can be constructed as follows:

$$\dfrac{\dfrac{\dfrac{\{this \rightarrow sdyn_{\mathsf{N}}(\mathsf{N}) * this \rightarrow Sok_{\mathsf{N}}()\}\_\{\mathtt{true}\}}{\left\{\begin{array}{l} this_{\mathsf{S}} \rightarrow sdyn_{\mathsf{S}}(\mathsf{N}) * this_{\mathsf{T}} \rightarrow tdyn_{\mathsf{T}}(\mathsf{N}) \\ \quad * this_{\mathsf{S}} \rightarrow Sok_{\mathsf{S}}() * this_{\mathsf{T}} \rightarrow Tok_{\mathsf{T}}() \end{array}\right\}\_\{\mathtt{true}\}}\ \textsc{APredDef}}{\{this \rightarrow tdyn_{\mathsf{N}}(\mathsf{N}) * this \rightarrow Tok_{\mathsf{N}}()\}\_\{\mathtt{true}\}}\ \textsc{APredDef}}{\{this_{\mathsf{T}} \rightarrow tdyn_{\mathsf{N}}(\mathsf{N}) * this_{\mathsf{T}} \rightarrow Tok_{\mathsf{N}}()\}\_\{\mathtt{true}\}}\ \textsc{AMovePred}$$

The behavioral suptyping proofs for the specifications of `setSource` and `setTarget`, and the proof that the specification of `~N` implies the specification of `~S`, can be established trivially using assertion weakening rule AMovePred.

5 RELATED WORK

Parkinson and Bierman's work [9, 10] introduces abstract predicate families. Their proof system allows a derived class to extend a base class, restrict the behavior of its base class, and alter the behavior of the base class while preserving behavioral subtyping. Method specifications consist of a dynamic and static specification, used for dynamically and statically dispatched calls, respectively. We derive these specifications from the same specification, using special variable θ. Their proof system only accounts for single inheritance without the presence of virtual destructors.

Ramananandro et al. [11] define operational semantics for a subset of C++, including construction and destruction in the presence of multiple inheritance and virtual methods that are dynamically dispatched. Their semantics encode the evolution of an object's dynamic type during construction and destruction. However, they only consider *stack-allocated* objects. This means that the concrete dynamic type of an object is always statically known at the point of its destruction.

Van Staden and Calcagno [14] extend the work of Parkinson and Bierman to a separation logic for object-oriented programs with multiple inheritance and virtual methods calls that are dynamically dispatched. They only consider virtual inheritance, which means that an object cannot have two base subobjects of the same class type. Furthermore, their logic does not support destructors, so polymorphic deletion is not considered. In their proof system, the dynamic type of an object is fixed after allocation, whereas we model the evolution of the dynamic type of an object during its construction and destruction.

BRiCk [13], built upon the separation logic of Iris [6], is a program logic for C++. The Frama-Clang plugin of Frama-C [7] enables analysis of C++ programs, supporting the ACSL specification language. Both tools support dynamic dispatching and model the evolution of an object's dynamic type through its construction and destruction. However, at the time of writing, no literature on these tools' approaches has appeared.

6 CONCLUSION

In this paper we proposed a separation logic for modular verification of programs where virtual method calls are bound to different methods at different points during the construction and destruction of objects. Additionally, we support polymorphic destruction where the static type of an object is a supertype of its dynamic type.

We defined the operational semantics of our language related to allocation and deallocation, construction and destruction, and method dispatching, and listed the corresponding proof rules for verification.

Next, we illustrated an example program annotated with a proof outline, to support our verification approach. This example indicates that our separation logic can be used to verify C++ dynamic binding in the presence of multiple inheritance. To our knowledge, we are the first to define a Hoare logic which reflects C++'s semantics of dynamic binding in the presence of constructors an destructors.

We implemented our approach [8] as part of our effort to extend our VeriFast tool for semi-automated modular formal verification of C and Java programs with support for C++. The implementation in VeriFast additionally supports bases that are non-polymorphic. One limitation is that our current operational semantics and separation logic does not consider virtual inheritance.

REFERENCES

[1] OpenJDK Community. 2023. OpenJDK. https://github.com/openjdk/jdk
[2] Francesco Dagnino. 2022. A Meta-Theory for Big-Step Semantics. *ACM Trans. Comput. Logic* 23, 3, Article 20 (apr 2022), 50 pages. https://doi.org/10.1145/3522729
[3] .NET Foundation and Contributors. 2023. .NET CLR. https://github.com/dotnet/runtime/tree/main/src/coreclr
[4] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259
[5] Google Inc., The Chromium Authors, The Chromium OS Authors, The Go Authors, and Samsung Inc. 2023. Fuchsia. https://github.com/vsrinivas/fuchsia
[6] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.
[7] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal aspects of computing* 27, 3 (2015), 573–609.
[8] Niels Mommen. 2023. VeriFast C++ tests. https://github.com/verifast/verifast/tree/master/tests/cxx

[9] Matthew Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. *SIGPLAN notices* 40, 1 (2005), 247–258.
[10] Matthew J. Parkinson and Gavin M. Bierman. 2008. Separation logic, abstraction and inheritance. *SIGPLAN notices* 43, 1 (2008), 75–86.
[11] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. 2012. A Mechanized Semantics for C++ Object Construction and Destruction, with Applications to Resource Management. *SIGPLAN notices* 47, 1 (2012), 521–532.
[12] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings - Symposium on Logic in Computer Science*. IEEE, Los Alamitos CA, 55–74.
[13] BedRock systems. 2023. BRiCk. https://github.com/bedrocksystems/BRiCk
[14] Stephan Van Staden and Cristiano Calcagno. 2009. Separation, abstraction, multiple inheritance and view shifting. *Technical report* 655 (2009).

A OPERATIONAL SEMANTICS

The operational semantics of expressions, commands, and constructor and destructor invocations are defined by mutual induction:

OLookup
$$\frac{h, e \Downarrow h' \uplus \{\![o \to f \mapsto v]\!\}, o}{h, e \to f \Downarrow h' \uplus \{\![o \to f \mapsto v]\!\}, v}$$

ODeleteNull
$$\frac{h, e \Downarrow h', \texttt{null}}{h, \texttt{delete}(e) \Downarrow h'}$$

OVal
$$h, v \Downarrow h, v$$

OUpdate
$$\frac{h, e \Downarrow h', o \qquad h', e' \Downarrow h'' \uplus \{\![o \to f \mapsto v]\!\}, v'}{h, e \to f := e' \Downarrow h'' \uplus \{\![o \to f \mapsto v']\!\}}$$

OLet
$$\frac{h, e \Downarrow h', v \qquad h', c[v/x] \Downarrow h''}{h, \texttt{let } x := e \texttt{ in } c \Downarrow h''}$$

OSeq
$$\frac{h, c \Downarrow h' \qquad h', c' \Downarrow h''}{h, c; c' \Downarrow h''}$$

OSkip
$$h, \texttt{skip} \Downarrow h$$

OUpcast
$$\frac{h, e \Downarrow h', o \qquad o :_{\texttt{st}} C \qquad C' \in bases(C)}{h, (C'*)\ e \Downarrow h', o_{C'}}$$

OStaticDispatch
$$\frac{\begin{array}{c}\texttt{class } C \cdots \{\cdots \texttt{virtual } m(\overline{x})\{c\} \cdots\} \qquad h, e \Downarrow h', o \\ o :_{\texttt{st}} C \qquad h', \overline{e} \Downarrow h'', \overline{v} \qquad h'', c[o/\texttt{this}, \overline{v}/\overline{x}] \Downarrow h'''\end{array}}{h, e \to C{::}m(\overline{e}) \Downarrow h'''}$$

ODynamicDispatch
$$\frac{\begin{array}{c}\texttt{class } C \cdots \{\cdots \texttt{virtual } m(\overline{x})\{c\} \cdots\} \\ h, e \Downarrow h', o \qquad h', \overline{e} \Downarrow h'', \overline{v} \\ dtype(o, C) \subseteq h'' \qquad o' = o \downarrow C \qquad h'', c[o'/\texttt{this}, \overline{v}/\overline{x}] \Downarrow h'''\end{array}}{h, e \to m(\overline{e}) \Downarrow h'''}$$

OConstruct
$$\frac{\begin{array}{c}\texttt{class } C : C_1 \ldots C_n \{\overline{f := \texttt{null};} \cdots C(\overline{x}) : C_1(\overline{e_1}) \ldots C_n(\overline{e_n})\{c\} \cdots\} \\ h, \overline{e} \Downarrow h_0, \overline{v} \\ h_0, o_{C_1} \to C_1(\overline{e_1}[o/\texttt{this}, \overline{v}/\overline{x}]) \Downarrow h_1 \uplus dtype(o_{C_1}, C_1) \\ \vdots \\ h_{n-1}, o_{C_n} \to C_n(\overline{e_n}[o/\texttt{this}, \overline{v}/\overline{x}]) \Downarrow h_n \uplus dtype(o_{C_n}, C_n) \\ h_n \uplus \{\![\overline{o \to f \mapsto \texttt{null}}]\!\} \uplus dtype(o, C), c[o/\texttt{this}, \overline{v}/\overline{x}] \Downarrow h'\end{array}}{h, o \to C(\overline{e}) \Downarrow h'}$$

ONew
$$\frac{\begin{array}{c} o = (id : C*) \\ \texttt{alloc}(id) \notin h \qquad h \uplus \{\![\texttt{alloc}(id)]\!\}, o \to C(\overline{e}) \Downarrow h'\end{array}}{h, \texttt{new } C(\overline{e}) \Downarrow h' \uplus \{\![\texttt{cted}(o, C)]\!\}, o}$$

ODestruct
$$\frac{\begin{array}{c}\texttt{class } C : C_1 \ldots C_n \{\overline{f := \texttt{null};} \cdots \texttt{virtual } \texttt{\~{}}C()\{c\} \cdots\} \\ h, c[o/\texttt{this}] \Downarrow h_n \uplus dtype(o, C) \uplus \{\![\overline{o \to f \mapsto v}]\!\} \\ h_n \uplus dtype(o_{C_n}, C_n), o_{C_n} \to \texttt{\~{}}C_n() \Downarrow h_{n-1} \\ \vdots \\ h_1 \uplus dtype(o_{C_1}, C_1), o_{C_1} \to \texttt{\~{}}C_1() \Downarrow h_0\end{array}}{h, o \to \texttt{\~{}}C() \Downarrow h_0}$$

ODelete
$$\frac{\begin{array}{c} o' = o \downarrow C \\ h, e \Downarrow h' \uplus \{\![\texttt{cted}(o', C)]\!\}, o \qquad h', o' \to \texttt{\~{}}C() \Downarrow h''\end{array}}{h, \texttt{delete}(e) \Downarrow h''}$$

B ASSERTION SEMANTICS

The semantics of assertions are defined as follows:

$$\begin{array}{lcl}
I, h \vDash b & \Leftrightarrow & b = \texttt{true} \\
I, h \vDash P * Q & \Leftrightarrow & \exists h_1, h_2 . h = h_1 \uplus h_2 \wedge \\
 & & \quad I, h_1 \vDash P \wedge I, h_2 \vDash Q \\
I, h \vDash P \wedge Q & \Leftrightarrow & I, h \vDash P \wedge I, h \vDash Q \\
I, h \vDash P \vee Q & \Leftrightarrow & I, h \vDash P \vee I, h \vDash Q \\
I, h \vDash \exists x.\ P & \Leftrightarrow & \exists v.\ I, h \vDash P[v/x] \\
I, h \vDash o \to p(C, \overline{v}) & \Leftrightarrow & \exists o'.\ o \downarrow C = o' \wedge (h, o', p, C, \overline{v}) \in I \\
I, h \vDash \texttt{cted}(o, C) & \Leftrightarrow & \exists o'.\ o \downarrow C = o' \wedge \texttt{cted}(o', C) \in h \\
I, h \vDash o :_{\texttt{dyn}} C & \Leftrightarrow & dtype(o, C) \subseteq h \\
I, h \vDash o \to f \mapsto v & \Leftrightarrow & o \to f \mapsto v \in h
\end{array}$$

where $I, h \vDash P$ means that assertion P is satisfied, given heap h and interpretation of predicates I. Cases not listed are false.

C PROOF RULES

We define evaluation contexts for expressions and commands as follows:

$$\begin{array}{rl}
K_e & ::= \bullet \mid K_e \to f \mid \texttt{new } C(\overline{v}\ K_e\ \overline{e}) \mid (C*)K_e \\
K_c & ::= \bullet \mid \texttt{delete } K_e \mid K_e \to f := e \mid o \to f := K_e \mid K_e \to C{::}m(\overline{e}) \\
 & \mid o \to C{::}m(\overline{v}\ K_e\ \overline{e}) \mid K_e \to m(\overline{e}) \mid o \to m(\overline{v}\ K_e\ \overline{e})
\end{array}$$

We use the notation $K[e]$ to denote the context K with expression e substituted for the hole $\bullet$.

HFrame
$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

HConseq
$$\frac{P \Rightarrow_a P' \qquad \{P'\}\ c\ \{Q'\} \qquad Q' \Rightarrow_a Q}{\{P\}\ c\ \{Q\}}$$

HNull
$$\{\texttt{true}\}\ \texttt{null}\ \{\texttt{result} = \texttt{null}\}$$

HPointer
$$\{\texttt{true}\}\ o\ \{\texttt{result} = o\}$$

HLookup
$$\{o \to f \mapsto v\}\ o \to f\ \{o \to f \mapsto v \wedge \texttt{result} = v\}$$

HUpdate
$$\{o \to f \mapsto \_\}\ o \to f := v\ \{o \to f \mapsto v\}$$

HLet
$$\frac{\{P\}\ e\ \{Q\} \qquad \forall v.\ \{Q[v/\texttt{result}]\}\ c[v/x]\ \{R\}}{\{P\}\ \texttt{let } x := e \texttt{ in } c\ \{R\}}$$

HSeq
$$\frac{\{P\}\ c\ \{Q\} \qquad \{Q\}\ c'\ \{R\}}{\{P\}\ c; c'\ \{R\}}$$

HSkip
$$\{P\}\ \texttt{skip}\ \{P\}$$

HContext
$$\frac{\{P\}\ e\ \{Q\} \qquad \forall v.\ \{Q[v/\texttt{result}]\}\ K[v]\ \{R\}}{\{P\}\ K[e]\ \{R\}}$$

HConsContext
$$\frac{\{P\}\ e\ \{Q\} \qquad \forall v.\ \{Q[v/\texttt{result}]\}\ o \longrightarrow C(\overline{v}\ v\ \overline{e})\ \{R\}}{\{P\}\ o \longrightarrow C(\overline{v}\ e\ \overline{e})\ \{R\}}$$

HConstruct
$$\frac{\texttt{class}\ C \cdots \{\cdots C(\overline{x})\ \texttt{req}\ P\ \texttt{ens}\ Q \cdots\} \in \Gamma}{\{P[\overline{v}/\overline{x}])\}\ o \longrightarrow C(\overline{v})\ \{Q[\overline{v}/\overline{x}, o/\texttt{this}]\}}$$

HNew
$$\frac{\texttt{class}\ C \cdots \{\cdots C(\overline{x})\ \texttt{req}\ P\ \texttt{ens}\ Q \cdots\} \in \Gamma}{\{P[\overline{v}/\overline{x}]\}\ \texttt{new}\ C(\overline{v})\ \{Q[\overline{v}/\overline{x}, \texttt{result}/\texttt{this}] * \texttt{cted}(\texttt{result}, C)\}}$$

HDestruct
$$\frac{\texttt{class}\ C \cdots \{\cdots \texttt{virtual}\ \ \texttt{\~{}}C()\ \texttt{req}\ P\ \texttt{ens}\ Q \cdots\} \in \Gamma}{\{P[o/\texttt{this}, C/\theta]\}\ o \longrightarrow \texttt{\~{}}C()\ \{Q\}}$$

HDeleteNull
$$\{\texttt{true}\}\ \texttt{delete(null)}\ \{\texttt{true}\}$$

HDelete
$$\frac{\texttt{class}\ C \cdots \{\cdots \texttt{virtual}\ \ \texttt{\~{}}C()\ \texttt{req}\ P\ \texttt{ens}\ Q \cdots\} \in \Gamma \qquad o :_{\texttt{st}} C}{\{\texttt{cted}(o, C') * P[o/\texttt{this}, C'/\theta]\}\ \texttt{delete}(o)\ \{Q\}}$$

HStaticDispatch
$$\frac{\texttt{class}\ C \cdots \{\cdots \texttt{virtual}\ m(\overline{x})\ \texttt{req}\ P\ \texttt{ens}\ Q \cdots\} \in \Gamma \qquad o :_{\texttt{st}} C}{\{P[o/\texttt{this}, C/\theta, \overline{v}/\overline{x}]\}\ o \longrightarrow C{::}m(\overline{v})\ \{Q[o/\texttt{this}, C/\theta, \overline{v}/\overline{x}]\}}$$

HDynamicDispatch
$$\frac{\texttt{class}\ C \cdots \{\cdots \texttt{virtual}\ m(\overline{x})\ \texttt{req}\ P\ \texttt{ens}\ Q \cdots\} \in \Gamma \qquad o :_{\texttt{st}} C}{\{o :_{\texttt{dyn}} C' \wedge P[o/\texttt{this}, C'/\theta, \overline{v}/\overline{x}]\}\ o \longrightarrow m(\overline{v})\ \{Q[o/\texttt{this}, C'/\theta, \overline{v}/\overline{x}]\}}$$

HExists
$$\frac{\forall v.\ \{P[v/x]\}\ c\ \{Q\}}{\{\exists x.\ P\}\ c\ \{Q\}}$$

HUpcast
$$\frac{o :_{\texttt{st}} C \qquad C \in bases(C')}{\{P[o_C/\texttt{result}]\}\ (C*)\ o\ \{P\}}$$

C.1 Destructor override check

$$\frac{\begin{array}{c}\texttt{class}\ C \cdots \{\cdots \texttt{virtual}\ \texttt{\~{}}C()\ \texttt{req}\ P\ \texttt{ens}\ Q \cdots\} \in \Gamma \\ \texttt{class}\ C' \cdots \{\cdots \texttt{virtual}\ \texttt{\~{}}C'()\ \texttt{req}\ P'\ \texttt{ens}\ Q' \cdots\} \in \Gamma \\ \Gamma \vdash \{P\}\_\{Q\}\ {}^{C}\!\!\Rightarrow^{C'}\ \{P'\}\_\{Q'\} \\ \forall C'' \in bases(C').\ \Gamma \vdash \text{override of destructor in } C'' \text{ correct in C}\end{array}}{\Gamma \vdash \text{override of destructor in } C' \text{ correct in } C}$$

C.2 Class verification

$$\frac{\begin{array}{c} class = \texttt{class}\ C \cdots \{\cdots ctor\ dtor\ \overline{meth}\ \} \\ \Gamma \vdash ctor \text{ correct in } C \\ \Gamma \vdash dtor \text{ correct in } C \qquad \Gamma \vdash \overline{meth} \text{ correct in } C\end{array}}{\Gamma \vdash class \text{ correct}}$$

D SOUNDNESS

Due to the fact that our assertion language does not allow predicate assertions in negative positions (i.e. under negation or on the left-hand side of implication), we have the following property:

Lemma 1. *The semantics of assertions is monotonic in the predicate interpretation I:*

$$I \subseteq I' \wedge I, h \vDash P \Rightarrow I', h \vDash P$$

Proof. By induction on the structure of P. □

We define a function F on predicate interpretations as follows:

$$F(I) = \left\{ (h, o, p, C, \overline{v}) \;\middle|\; \begin{array}{l} \texttt{class}\ C\ \cdots\ \{\ \cdots\ \texttt{pred}\ p(\overline{x}) = P;\ \cdots\} \\ \wedge I, h \vDash P[o/\texttt{this}, \overline{v}/\overline{x}] \end{array} \right\}$$

We define the program's predicate interpretation I_{program} by $I_{\text{program}} = \bigcap\{I \mid F(I) \subseteq I\}$. By the Knaster-Tarski theorem, I_{program} is a fixpoint of F: $F(I_{\text{program}}) = I_{\text{program}}$.[4] We use notation $h \vDash P$ to mean $I_{\text{program}}, h \vDash P$.

Lemma 2 (Soundness of assertion weakening).

$$P \Rightarrow_a Q \wedge h \vDash P \Rightarrow h \vDash Q$$

Proof. By induction on the derivation of $P \Rightarrow_a Q$. □

We define semantic counterparts of the correctness judgments of our proof system as follows:

$$\vDash \{P\}\ e\ \{Q\} \Leftrightarrow \left(\forall h, h_{\text{f}}.\ h \vDash P \Rightarrow \begin{array}{l} h \uplus h_{\text{f}}, e\ \texttt{div}\ \vee \\ \exists h', v.\ h \uplus h_{\text{f}}, e \Downarrow h' \uplus h_{\text{f}}, v \\ \qquad \wedge h' \vDash Q[v/\texttt{result}] \end{array} \right)$$

$$\vDash \{P\}\ c\ \{Q\} \Leftrightarrow \left(\forall h, h_{\text{f}}.\ h \vDash P \Rightarrow \begin{array}{l} h \uplus h_{\text{f}}, c\ \texttt{div}\ \vee \\ \exists h'.\ h \uplus h_{\text{f}}, c \Downarrow h' \uplus h_{\text{f}} \wedge h' \vDash Q \end{array} \right)$$

$$\vDash \{P\}\ o \longrightarrow C(\overline{e})\ \{Q\} \Leftrightarrow \left(\forall h, h_{\text{f}}.\ h \vDash P \Rightarrow \begin{array}{l} h \uplus h_{\text{f}}, o \longrightarrow C(\overline{e})\ \texttt{div}\ \vee \\ \exists h'.\ h \uplus h_{\text{f}}, o \longrightarrow C(\overline{e}) \Downarrow h' \uplus h_{\text{f}} \wedge h' \vDash Q \end{array} \right)$$

$$\vDash \{P\}\ o \longrightarrow \texttt{\~{}}C()\ \{Q\} \Leftrightarrow \left(\forall h, h_{\text{f}}.\ h \vDash P \Rightarrow \begin{array}{l} h \uplus h_{\text{f}}, o \longrightarrow \texttt{\~{}}C()\ \texttt{div}\ \vee \\ \exists h'.\ h \uplus h_{\text{f}}, o \longrightarrow \texttt{\~{}}C() \Downarrow h' \uplus h_{\text{f}} \wedge h' \vDash Q \end{array} \right)$$

Lemma 3. *Soundness of HContext If* $\vDash \{P\}\ e\ \{Q\}$ *and* $\forall v.\ \vDash \{Q[v/\texttt{result}]\}\ K[v]\ \{R\}$ *then* $\vDash \{P\}\ K[e]\ \{R\}$.

Proof. By induction on the structure of K. □

Assumption 1. *The program is correct:*

$$\vdash \textit{program correct}$$

[4] It is in fact the least fixpoint.

Lemma 4 (Main Soundness Lemma).

$$\begin{array}{l}\forall h, h_f, P, Q.\ h \vDash P \Rightarrow \\ \quad (\forall e.\ \{P\}\ e\ \{Q\} \wedge \\ \quad\quad (\nexists h', v.\ h \uplus h_f, e \Downarrow h' \uplus h_f, v \wedge h' \vDash Q[v/\mathtt{result}]) \Rightarrow \\ \quad\quad\quad h \uplus h_f, e\ \mathtt{div}) \wedge \\ \quad (\forall c.\ \{P\}\ c\ \{Q\} \wedge (\nexists h'.\ h \uplus h_f, c \Downarrow h' \uplus h_f \wedge h' \vDash Q) \Rightarrow \\ \quad\quad h \uplus h_f, c\ \mathtt{div}) \wedge \\ \quad (\forall o, C, \overline{e}.\ \{P\}\ o \rightarrow C(\overline{e})\ \{Q\} \wedge \\ \quad\quad (\nexists h'.\ h \uplus h_f, o \rightarrow C(\overline{e}) \Downarrow h' \uplus h_f \wedge h' \vDash Q) \Rightarrow \\ \quad\quad\quad h \uplus h_f, o \rightarrow C(\overline{e})\ \mathtt{div}) \wedge \\ \quad (\forall o, C.\ \{P\}\ o \rightarrow \text{\textasciitilde}C()\ \{Q\} \wedge \\ \quad\quad (\nexists h'.\ h \uplus h_f, o \rightarrow C(\overline{e}) \Downarrow h' \uplus h_f \wedge h' \vDash Q) \Rightarrow \\ \quad\quad\quad h \uplus h_f, o \rightarrow \text{\textasciitilde}C()\ \mathtt{div})\end{array}$$

Proof. By mutual co-induction and, nested inside of it, induction on the derivation of the correctness judgment. We elaborate a few cases:

- Case HDynamicDispatch. Assume the following:

$$\begin{array}{c} c = o \rightarrow m(\overline{v}) \\ o :_{\mathsf{st}} C \\ \mathtt{class}\ C \cdots \{ \cdots \mathtt{virtual}\ m(\overline{x})\ \mathtt{req}\ P_C\ \mathtt{ens}\ Q_C \cdots \} \\ P = o :_{\mathsf{dyn}} D \wedge P_C[o/\mathtt{this}, D/\theta, \overline{v}/\overline{x}] \\ Q = Q_C[o/\mathtt{this}, D/\theta, \overline{v}/\overline{x}] \\ \mathtt{class}\ D \cdots \{ \cdots \mathtt{virtual}\ m(\overline{x})\ \mathtt{req}\ P_D\ \mathtt{ens}\ Q_D\ \{c_m\} \cdots \} \end{array}$$

  By $h \vDash P$, we have $dtype(o, D) \subseteq h$ and $h \vDash P_C[o/\mathtt{this}, D/\theta, \overline{v}/\overline{x}]$. Let $o' = o \downarrow D$. By the correctness of method m in class D, we have

$$\begin{array}{c} \{P_D[o'/\mathtt{this}, D/\theta, \overline{v}/\overline{x}]\} \\ c_m[o'/\mathtt{this}, \overline{v}/\overline{x}] \\ \{Q_D[o'/\mathtt{this}, D/\theta, \overline{v}/\overline{x}]\} \end{array}$$

  By the fact that m in D correctly overrides m in C, we have

$$\{P_D\}\_\{Q_D\}\ {}^{D}{\Rightarrow}^{C}\ \{P_C\}\_\{Q_C\}$$

  It follows that

$$\begin{array}{c} \{P_C[o/\mathtt{this}, D/\theta, \overline{v}/\overline{x}]\} \\ c_m[o'/\mathtt{this}, \overline{v}/\overline{x}] \\ \{Q_C[o/\mathtt{this}, D/\theta, \overline{v}/\overline{x}]\} \end{array}$$

  The relevant inference rule for divergence of dynamically dispatched method calls is as follows:

  ODynamicDispatchDiv3

$$\frac{\begin{array}{c} \mathtt{class}\ C \cdots \{ \cdots \mathtt{virtual}\ m(\overline{x})\{c\} \cdots \} \\ h, e \Downarrow h', o \qquad o' = o \downarrow C \\ h', \overline{e} \Downarrow h'', \overline{v} \\ dtype(o, C) \subseteq h'' \qquad h'', c[o'/\mathtt{this}, \overline{v}/\overline{x}]\ \mathtt{div} \end{array}}{h, e \rightarrow m(\overline{e})\ \mathtt{div}}$$

  We apply this rule to the goal, which reduces the goal to $h, c_m[o'/\mathtt{this}, \overline{v}/\overline{x}]$ div. We now apply the coinduction hypothesis. We are now left with the job of proving that the body does not terminate, assuming that the call does not terminate. Instead, we prove that the call terminates, assuming that the body terminates. We conclude that proof by applying ODynamicDispatch.

- Case HConsContext. Assume a constructor argument list $\overline{v}\, e\, \overline{e}$. By the induction hypothesis corresponding to the first premise of HConsContext, we have that evaluation of e either terminates or diverges.
  - Assume e terminates with a value v. By the induction hypothesis corresponding to the second premise of HConsContext, we have that $o \rightarrow C(\overline{v}\, v\, \overline{e})$ either terminates or diverges.
    - Assume $o \rightarrow C(\overline{v}\, v\, \overline{e})$ terminates. This must be by an application of OConstruct. Therefore, it must be that $\overline{e}$ all terminate. It follows that $o \rightarrow C(\overline{v}\, e\, \overline{e})$ terminates.
    - Assume $o \rightarrow C(\overline{v}\, v\, \overline{e})$ diverges. Given that e terminates, we can easily prove that $o \rightarrow C(\overline{v}\, e\, \overline{e})$ diverges.
  - Assume e diverges. Then $o \rightarrow C(\overline{v}, e, \overline{e})$ diverges.
- Case HContext. We apply Lemma 3 and use the induction hypotheses to discharge the resulting subgoals.[5]

□

[5] To see that this preserves productivity of the coinductive proof, notice that Lemma 3 is *size-preserving*: given approximations up to depth d of the proof trees for the lemma's premises, the lemma produces a proof tree of depth at least d.