A mechanized semantics for C++ object construction and destruction, with applications to resource management

Tahina Ramananandro, Gabriel Dos Reis, Xavier Leroy

A Mechanized Semantics for C++ Object Construction and Destruction, with Applications to Resource Management

Tahina Ramananandro
INRIA Paris-Rocquencourt
tahina.ramananandro@inria.fr

Gabriel Dos Reis *
Texas A&M University
gdr@cs.tamu.edu

Xavier Leroy †
INRIA Paris-Rocquencourt
xavier.leroy@inria.fr

Abstract

We present a formal operational semantics and its Coq mechanization for the C++ object model, featuring object construction and destruction, shared and repeated multiple inheritance, and virtual function call dispatch. These are key C++ language features for high-level system programming, in particular for predictable and reliable resource management. This paper is the first to present a formal mechanized account of the metatheory of construction and destruction in C++, and applications to popular programming techniques such as "resource acquisition is initialization." We also report on irregularities and apparent contradictions in the ISO C++03 and C++11 standards.

Categories and Subject Descriptors D.1.5 [*Programming techniques*]: Object-oriented Programming; D.2.0 [*Software Engineering*]: General—*Standards*; D.2.2 [*Software Engineering*]: Design Tools and Techniques—*Object-oriented design methods*; D.2.4 [*Software Engineering*]: Software/Program Verification—*Correctness proofs*; D.3.3 [*Programming Languages*]: Language Constructs and Features—*Classes and objects*, *Inheritance*; F.3.3 [*Logics and meanings of programs*]: Studies of program constructs—*Object-oriented constructs*

General Terms Languages, Verification

1. Introduction

One of the earliest decisions for C++ (in 1979) was to provide language support for the construction and destruction of objects [15]: programmer-supplied code fragments called *constructors* are automatically executed when a new object is created and before it is made available to the rest of the program; these constructors can execute arbitrary code sequences and are intended to establish the execution environment for operations by initializing fields and maintaining class-level invariants. Symmetrically, the language also supports *destructors*, which are executed before an object is destroyed at precisely-defined times. Such general constructors are now available in most programming languages supporting object-oriented programming. Many languages also provides some support for clean-up of objects at the end of their lifetime, e.g. finalizers executed asynchronously at loosely-specified times or `dispose` statements allowing a programmer to explicitly request clean-up.

Because it is enforced by the programming language itself and not by coding conventions, the object construction and destruction protocol provides strong guarantees that can be leveraged by good programming practices. This is the case for *Resource Acquisition Is Initialization* (RAII), a popular C++ programming idiom for safely managing system resources such as file descriptors. With RAII, resource acquisition is always performed in constructors, and resource release in the corresponding destructors. Since C++ guarantees that every execution of a constructor is eventually matched by an execution of a destructor, RAII minimizes the risk of leaking resources.

As practically useful as they are, constructors and destructors raise delicate issues for the programmers, for static analysis tools, and even for language designers. While an object is being constructed or destructed, it is not in a fully operational state. This raises the question of what operations constructor/destructor code should be allowed to perform on this object, and what semantics to give to these operations. A typical example of this dilemma is virtual method dispatch: in Java and C#, unrestricted virtual method calls are allowed in constructors, with the same semantics as on fully constructed objects; in contrast, C++ takes object construction states into account when resolving virtual function calls, in a way that provides stronger soundness guarantees but complicates semantics and compilation. Section 2 reviews the main features of C++ object construction and destruction.

This paper reports on a formal, mechanized semantics for construction and destruction in the C++ object model. This model is especially rich: it features multiple inheritance (both shared and repeated), fields of structure and structure array types, as well as subtle interactions between construction states, virtual function calls, and dynamic casts. Our semantics, fully mechanized in the Coq proof assistant, is, to the best of our knowledge, the first to capture faithfully these interactions as intended by the C++ standard. This semantics, described in section 3, is the first contribution of the paper.

The second contribution, in section 4, is the specification and formal verification of several expected properties of the construction/destruction protocol, ranging from common-sense properties such as "any subobject is constructed, or destroyed, at most once" to the first precise characterization of the properties that make the RAII programming idiom work in practice. We also study the evolutions of the dynamic types of objects throughout their life cycle and formally justify the program points where compilers must generate code to update dynamic types (v-tables).

As a third contribution, this work exposes some irregularities and apparent contradictions in the ISO C++03 and C++11 stan-

* Partially supported by NSF grants CCF-1035058 and OISE-1043084.

† Partially supported by ANR grants Arpège U3CAT and INS Verasco.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

dards, summarized in section 5. These have been reported to the ISO C++ committee; some were resolved in C++11, others will be in future revisions. We finish the paper with a few words about verified compilation (§6), a discussion of related work (§7) and future work (§8), and concluding remarks (§9).

All results presented in this paper were mechanized using the Coq proof assistant [1]. The Coq development is available at `http://gallium.inria.fr/~tramanan/cxx/`. By lack of space, some aspects of the semantics are only sketched in this paper. A full description can be found in the first author's Ph.D. thesis [12, part 2].

2. Object construction and destruction in C++

A constructor is a special function that turns "raw memory" into an object. Its purpose is to create an environment where class invariants hold, and where the class's member functions will operate [2, 15, 16]. Conversely, at the end of the useful life of an object, another special function — called destructor — releases any acquired resources and turns the object back to mere raw memory. The time span between the completion of a constructor and the start of a destructor is the *lifetime* of the constructed object. The C++ language rules were designed to guarantee that class invariants set up by a constructor hold through the entire lifetime of an object. For example, a complete object's dynamic type does not change during its lifetime.

C++'s object construction and destruction mechanism is rooted in a few design principles:

I. Every object and each of its subobjects (if any) is constructed exactly once.
II. No object construction should rely on parts yet to be constructed, and no object destruction should rely on parts already destroyed.
III. Every object is destroyed in the exact reverse order it was constructed.

These principles, independently of implementation considerations, enable support for reliable, modular, large-scale software development.

Principle I supports the idea that a resource acquired by an object (say a file lock) is predictably acquired exactly once.

Principle II embodies a notion of locality and scoping. While the constructor is creating the computation environment for member functions, very few class guarantees can be made. It is desirable to localize the complexity and/or convolution needed to correctly call a function in a few specific places before an object begins its lifetime. Furthermore, the author of a class should not have to worry about which data members of possible derived classes are correctly initialized during the execution of a constructor. On the other hand, she should be able to rely on base-class invariants. Consider the following example:

```
struct Locus {
  Locus(int a, int b) : x(a), y(b) { show(); }
  virtual void show() const {
    cout << x << ' ' << y << endl;
  }
protected:
  int x;
  int y;
};

struct FeaturedLocus : Locus {
  FeaturedLocus(int a, int b, const vector& v)
    : Locus(a, b), features(v) { }
  void show() const {
    cout << x << ' ' << y
         << ' ' << features << endl;
  }
private:
  vector features;
};
```

Construction of a `FeaturedLocus` object entails constructing its `Locus` base-class subobject, followed by copy-initialization of its `features` field from the `v` parameter. If, in the `Locus` constructor, the call to the virtual function `show` resolved to the final overrider `FeaturedLocus::show` (as in Java or C#), it would access the yet-to-be-constructed field `FeaturedLocus::features` – an undesirable and unpleasant outcome. Consequently, Principle II argues for resolving the call (in the constructor) to an overrider not from a derived class; here it is `Locus::show`.

Finally, Principle III supports predictability and reliability. For instance, if an object acquires N resources through the construction of its subobjects, in general one would want to release them in the reverse order of acquisition, or else a deadlock might ensue.

Turning these general principles (and their implications) into executable specification in the face of object-oriented language features such as multiple inheritance (both repeated and shared), late binding function dispatch (virtual function calls), type queries, etc. is an interesting challenge. On the other hand, their fulfillment leads to interesting soundness results and, more importantly, reliable software development techniques for the programmer.

Initialization versus assignment There is a difference between initialization and "declaration followed by a first-time assignment". And it is not a stylistic distinction. Base-class subobjects and nonstatic data members are constructed before the proper body of the constructor is executed. Initializers for subobjects are specified in a comma-separated list before the opening brace of a constructor body (see the definition of the constructors for `Locus` and `FeaturedLocus` in the previous example.) In particular, there is *no other way* to initialize base-class subobjects, const or reference nonstatic data members. A base-class subobject, or a nonstatic data member, not mentioned in the member-initializer list is implicitly initialized with a default constructor, i.e. a constructor that can be called with no argument. Consequently, the language rules imply that all subobjects are constructed by the time the first statement of the constructor body (if not empty) is executed. The constructor for `FeaturedLocus` could have been written as

```
FeaturedLocus(int a, int b, const vector& v)
  : Locus(a,b) { features = v; }
```

In this variation, the `Locus` base-class subobject is initialized; the nonstatic data member `features` is default constructed, and finally the constructor body *assigns* the value of `v` to `features`. In C++, assignments of objects of class types are generally implemented by user-defined functions. These functions must assume that the left hand side of the assignment is an object *already constructed and in a valid state*. In particular, `vector`'s assignment function must carefully check for self-assignment to avoid premature memory release; it must properly release the resources the object was managing before taking hold on the new ones. By contrast, during copy-construction, the object being constructed has *not* acquired any resources yet; so the implementation code of the copy-constructor is usually far simpler than that for assignment.

Initialization order Principle III has an interesting implication on the construction order of the constituents of an object. Observe that there is exactly one destructor per class, and that there can be as many constructors (per class) as deemed necessary by the programmer. Therefore, for a given class, subobjects in all constructors must be initialized in the same order — for that order must be the reverse destruction order of the (unique) destructor. Any well-defined order can do. However, for simplicity, a natural choice is the declaration

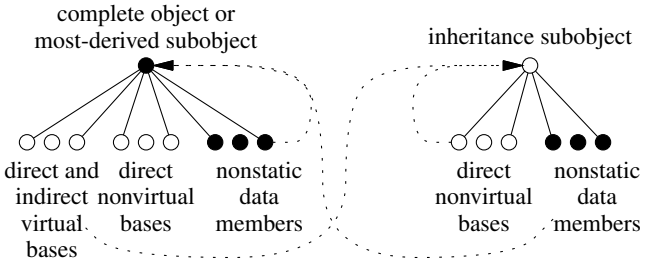

Figure 1. A recursive tree representation of the subobjects of a class, such that a depth-first left-to-right traversal yields the subobject construction order.

order: initialization of all base-class subobjects in the (left-to-right) order of declaration in the class definition; followed by initialization of all nonstatic data members in their order of declarations in the class definition.

This construction order is simple, predictable, and reflects the compositional aspect of "class growth". However, shared inheritance (virtual base classes) poses a problem for the principle of "base before derived" and "strict order of declaration construction". Indeed, consider the following hierarchy of IO stream classes:

```
struct istream : virtual basic_ios { ... };
struct ostream : virtual basic_ios { ... };
struct iostream : istream, ostream { ... };
```

where the class `basic_ios` maintains states and implements operations common to both input and output streams; the classes `istream` and `ostream` implement input and output stream functionalities. The class `iostream` combines both input and output facilities. If an `iostream` object was to be constructed according to the declaration order outlined above, the corresponding `basic_ios` base-class subobject would be constructed twice: once when the `istream` base-class subobject is constructed, and a second time when the `ostream` base-class subobject is constructed. This problem is resolved by executing constructors for virtual bases (in declaration order) before all other constructors. More generally, C++ retains the following *almost-declaration order* construction for a complete object, also depicted on Figure 1:

1. virtual base-class subobjects are constructed in a depth-first left-to-right traversal of the inheritance graph;
2. direct non-virtual base-class subobjects are constructed in declaration order;
3. nonstatic data members are constructed in declaration order.

Builtin object types The C++ programming language was designed with two competing ideals: a sound high-level object model with equal support for user-defined types and builtin types, and a near-total compatibility with C for system programming. The resulting tension has made the general model of object construction and destruction grow barnacles. Indeed, C lacks the notion of object construction so central to C++'s object model. But, in reality, both languages agree on the semantics of programs that read from uninitialized objects: undefined behavior, except in very limited cases where a read access is done as "raw memory". In C, object initialization is usually expressed in terms of storage acquisition followed by assignment; C++ distinguishes assignment from construction. To achieve the same ultimate semantics and to support generic programming, C++ extended the notion of constructor (and destructor) to builtin types. A builtin data member can be explicitly initialized with default value using the default construction syntax:

```
struct Locus { Locus() : x(), y() { } };
```

Here, the default constructor `Locus` explicitly constructs the data members `x` and `y` with a default value (zero) so that in the complete object declaration "`Locus origin;`", the `origin` object has its coordinates initialized to zero. Would the semantics be the same were `x` and `y` not explicitly listed in the member-initializer list? The answer is no: objects of builtin type with no explicit initializers are constructed but, left with indeterminate values. This irregularity was introduced as a way to satisfy the widespread practice in C where objects are given first-time values long after their declarations. This irregularity, and others related to destructors for builtin types, surely complicate the C++ semantics rules as well as formalization. However, they are not fundamental to the object model. In fact, the C++ standards committee is considering unifying initialization and lifetime rules in a post-C++11 standard. We hope this work will help in that endeavor.

3. Formal operational semantics

3.1 Syntax of the core language

We focus on a core language of objects that features the main aspects of the C++ object model: construction and destruction; multiple inheritance (both virtual and non-virtual), virtual functions, and nonstatic data members (a.k.a. "fields") of scalar, array or complete non-abstract object types. To simplify the presentation, we formalize only fields that are arrays of object types, viewing a field of object type T as a one-element array of type "$T[1]$".

The core language is a language of statements operating over variables in 3-address style. We assume that typechecking was performed, static overloading was resolved, and expressions were decomposed into elementary statements. The syntax of statements follows.

$op, \ldots$	: Op	Builtin operations
$var, \ldots$	: Var	Variables
$B, C, \ldots$	: $Class$	Classes
$fname$	: $Field$	Field names
$mname$	: $Method$	Method names
$Stmt$	::= skip	Do nothing
	$\mid var' := op(var^*)$	Builtin operation
	$\mid var' := var\text{->}_C fname$	Field read
	$\mid var\text{->}_C fname := var'$	Scalar field write
	$\mid var' := \&var[var_{\mathsf{index}}]_C$	Array cell access
	$\mid var' :=$ $\mathtt{dynamic\_cast}\langle B\rangle_C(var)$	Dynamic cast
	$\mid var'\text{->}_C mname(var^*)$	Virtual function call
	$\mid Stmt_1; Stmt_2$	Statement sequence
	$\mid \mathsf{return}$	Function return
	$\mid \{C\ var[var_{\mathsf{count}}] =$ $\{ObjInit^*\}; Stmt\}$	Block-scoped object

The Coq formalization also features simple control structures (if/then/else, infinite loops with early exits), as well as static casts. We leave these extra features out in this paper to concentrate on the essential features of the language: accesses to fields; virtual function calls; dynamic casts; and block-scoped objects. The statement $\{C\ x[N] = \{ObjInit^*\}; Stmt\}$ allocates an array of N objects of type C, constructs each of its elements according to the list of initializers $ObjInit^*$, binds the array to variable x, executes the block body $Stmt$, destructs each element of the array x, and finally deallocates the whole array.

Initializers are syntactically presented in C++ as constructor calls with expressions as arguments. Since our language lacks expressions, we model initializers as a statement (which evaluates the expression arguments into temporary variables) followed by an invocation of a constructor for object fields or the initialization of a scalar field.

$$
\begin{array}{rll}
\mathit{ObjInit} & ::= & \mathit{Stmt}; C(\mathit{var}^*) \quad \text{Class object initializer} \\
\mathit{FieldInit} & ::= & (\mathit{Stmt}; \mathit{fname}(\mathit{var})) \quad \text{Scalar field initializer} \\
& | & \mathit{fname}\{\mathit{ObjInit}^*\} \quad \text{Structure field initializers (one for each array cell)} \\
\mathit{Init} & ::= & \mathit{ObjInit} \mid \mathit{FieldInit}
\end{array}
$$

Arguments to functions and constructors can be scalar values, i.e. integers, or floating point data, or pointers to objects. We do not model passing objects by value: this raises delicate issues with the construction and destruction of *temporary objects*, discussed in §8.

$$
\begin{array}{rll}
\mathit{Constr} & ::= & C(\mathit{var}^*) : \mathit{Init}^*\{\mathit{Stmt}\} \quad \text{Constructor} \\
\mathit{Destr} & ::= & \sim C()\{\mathit{Stmt}\} \quad \text{Destructor} \\
\mathit{MethodDef} & ::= & \mathsf{virtual\ void} \quad \text{Virtual function} \\
& & \mathit{mname}(\mathit{var}^*)\{\mathit{Stmt}\} \quad \text{(method)} \\
\mathit{FieldDef} & ::= & \mathsf{scalar}\ \mathit{fname}; \quad \text{Data member} \\
& | & \mathsf{struct}\ C[\mathit{size}]\ \mathit{fname}; \quad \text{(field)} \\
\mathit{Base} & ::= & B \mid \mathsf{virtual}\ B \\
\mathit{ClassDef} & ::= & \mathsf{struct}\ C : \mathit{Base}^* \\
& & \{\mathit{FieldDef}^*\ \mathit{MethodDef}^* \\
& & \mathit{Constr}^*\ \mathit{Destr}\} \quad \text{Class definition} \\
\mathit{Program} & ::= & \mathit{ClassDef}^*; \\
& & \mathsf{main}()\{\mathit{Stmt}\} \quad \text{Program}
\end{array}
$$

A complete program consists of a hierarchy of classes. Each class definition contains a list of virtual and non-virtual base classes, a list of data members (a.k.a. fields), definitions for virtual functions (a.k.a. methods), one or several constructors, and one destructor. Each constructor consists of a parameter list, a sequence of initializers (for direct non-virtual bases, fields, and direct or indirect virtual bases), and a constructor body (an arbitrary statement). Destructors take no parameters and consist only of a body.

3.2 Designating subobjects

A crucial issue in formalizing the C++ object model is to capture the fact that each object of a class C contains *subobjects*, one for each base and each field of C. Subobjects can be arbitrarily nested, but can also be shared along several inheritance paths, owing to virtual inheritance. For this reason, a subobject cannot be identified by a store locations ℓ; instead, it is described by a pair of the location ℓ of the array of complete objects containing it, and a *path* from this array to the desired subobject. Intuitively, a path is a sequence of selection operations of the form "select a base class" or "select a field" or "select an element of an array". This path-based approach was introduced by Rossie and Friedman [14], later mechanized in Isabelle/HOL by Wasserrab *et al.* [18], and further extended with structure array fields and mechanized in Coq by Ramananandro *et al.* [13]. We now briefly recall this path-based formalization, referring the reader to [13, 18] for full details.

An *inheritance path* σ is a pair (h, l) of an inheritance kind $h ::= \mathsf{Repeated} \mid \mathsf{Shared}$ and a list of class names l. Such a path designates a *base-class subobject* of some type A of an object of type C. If $h = \mathsf{Repeated}$, it is a path through the *repeated*, or *non-virtual*, inheritance graph. Each path from C to A corresponds to a distinct copy of A within C. If $h = \mathsf{Shared}$, it is a path from a virtual base B of C to A, regardless of how the virtual base B is reached.

Not all paths are valid with respect to the class hierarchy. The following inference rules define the relation $C \dashv\langle(h,l)\rangle^{\mathcal{I}}_{\mapsto} A$, meaning that (h, l) is a valid inheritance path from C to a base-class subobject of C of static type A.

$$C \dashv\langle(\mathsf{Repeated}, C :: \epsilon)\rangle^{\mathcal{I}}_{\mapsto} C$$

$$\frac{B \text{ direct non-virtual base of } C \qquad B \dashv\langle(\mathsf{Repeated}, l)\rangle^{\mathcal{I}}_{\mapsto} A}{C \dashv\langle(\mathsf{Repeated}, C :: l)\rangle^{\mathcal{I}}_{\mapsto} A}$$

$$\frac{B \text{ virtual base of } C \qquad B \dashv\langle(h,l)\rangle^{\mathcal{I}}_{\mapsto} A}{C \dashv\langle(\mathsf{Shared}, l)\rangle^{\mathcal{I}}_{\mapsto} A}$$

An object is said to be a *most-derived* object if, and only if, it is not a base-class subobject of any other object. A most-derived object of type C is designated by the trivial inheritance path $(\mathsf{Repeated}, C :: \epsilon)$.

Inheritance paths compose naturally: if $C \dashv\langle(h,l)\rangle^{\mathcal{I}}_{\mapsto} B$ and $B \dashv\langle(h',l')\rangle^{\mathcal{I}}_{\mapsto} A$, we have $C \dashv\langle(h,l)@(h',l')\rangle^{\mathcal{I}}_{\mapsto} A$. Here, $@$ is the *cast to base* operator, defined as follows: for a cast through non-virtual inheritance, $(h,l)@(\mathsf{Repeated}, B :: l') = (h, l{+}{+}l')$; whereas through virtual inheritance, we have $(h,l)@(\mathsf{Shared}, l') = (\mathsf{Shared}, l')$. (We write $+{+}$ for list concatenation.)

In the presence of fields that are structures or arrays of structures, the notions of paths and subobjects must be extended to allow selecting elements of arrays of structures. Ramananandro *et al.* [13] define *array paths* as lists of (i, σ, F) triples, where i is an array index, σ an inheritance path, and F the name of a structure array field. An array path goes from an array of n structures of type C to an array of n' structures of type C', which we write $C[n] \dashv\langle\alpha\rangle^{\mathcal{A}}_{\mapsto} C'[n']$ and define by the inference rules below. An auxiliary predicate $C[n] \dashv\langle i, \sigma\rangle^{\mathcal{CI}}_{\mapsto} A$ captures the selection of the i-th element of an array followed by the extraction of a base-class subobject σ of type A.

$$\frac{0 \le i < n \qquad C \dashv\langle\sigma\rangle^{\mathcal{I}}_{\mapsto} A}{C[n] \dashv\langle i, \sigma\rangle^{\mathcal{CI}}_{\mapsto} A} \qquad \frac{0 \le n' \le n}{C[n] \dashv\langle\epsilon\rangle^{\mathcal{A}}_{\mapsto} C[n']}$$

$$\frac{F = (f, D, m) \text{ is a structure field defined in } A \qquad C[n] \dashv\langle i, \sigma\rangle^{\mathcal{CI}}_{\mapsto} A \qquad D[m] \dashv\langle\alpha\rangle^{\mathcal{A}}_{\mapsto} C'[n']}{C[n] \dashv\langle(i, \sigma, F) :: \alpha\rangle^{\mathcal{A}}_{\mapsto} C'[n']}$$

In the core language of §3.1, a *complete* object (bound to a variable x by a block statement) is always an array of structures, with type $C[n]$. A *subobject* of type A of such a complete object is. therefore, a triple (α, i, σ) where α is an array path from $C[n]$ to some $C'[n']$ and $i \in [0, n')$ is an index in the array of type $C'[n']$ and σ is the inheritance path for a base-class subobject of C' of type A. We write $C[n] \dashv\langle(\alpha, i, \sigma)\rangle\mapsto A$ to mean that (α, i, σ) designates a valid subobject of type A of the array $C[n]$. This relation is defined by:

$$\frac{C[n] \dashv\langle\alpha\rangle^{\mathcal{A}}_{\mapsto} C'[n'] \dashv\langle i, \sigma\rangle^{\mathcal{CI}}_{\mapsto} A}{C[n] \dashv\langle(\alpha, i, \sigma)\rangle\mapsto A}$$

Then, a subobject of a complete C object is a most-derived C' object if, and only if, it can be written as $(\alpha, i, (\mathsf{Repeated}, C' :: \epsilon))$ where $C[n] \dashv\langle\alpha\rangle^{\mathcal{I}}_{\mapsto} C'[n']$ and $0 \le i < n'$. In practice, this means that a most-derived object corresponds to a cell of a structure array, the array being either the complete C object itself ($\alpha = \epsilon$) or some structure field of a subobject of the complete object.

Our operational semantics uses these notions to unambiguously identify complete objects by locations ℓ in the store, subobjects by pairs $(\ell, (\alpha, i, \sigma))$ of a location and a subobject, and fields by triples $(\ell, (\alpha, i, \sigma), f)$ of a subobject of some type A and a name f of a field declared in A.

3.3 Construction states

In the C++ language, the behaviors of operations over objects depend on the *construction state* of these objects. For example, it is not possible to invoke a virtual function of a class C if the base classes of C have not been constructed yet, or are undergoing

destruction. Likewise, as described in §2, virtual method dispatch behaves differently when a most-derived object is fully constructed and when it is undergoing construction or destruction.

Our operational semantics therefore associates a construction state to every subobject and every field, and updates these states during construction and destruction steps. A given subobject can be in one of 7 construction states, whose meanings differ slightly depending on whether the subobject is a most-derived subobject (e.g. an element of a structure array) or a base-class subobject. For a most-derived object, the construction states and their meanings are:

1. Unconstructed: Construction has not started yet.
2. StartedConstructing: The construction of base-class subobjects has started, but not the fields.
3. BasesConstructed: The base-class subobjects are completely constructed. Now constructing the fields or executing the constructor body.
4. Constructed: The constructor body has returned, and destruction has not started yet.
5. StartedDestructing: The body of the destructor is executing or the fields are undergoing destruction.
6. DestructingBases: The fields have been completely destructed. Bases are undergoing destruction.
7. Destructed: All bases and fields have been destructed.

For a base-class subobject, we need to exclude virtual bases from the meaning of its construction state. Indeed, as discussed in §2 and shown in Figure 1, virtual bases are morally attached to the enclosing most-derived object, not to the base-class subobject. Therefore, for those base class subobjects that are not most-derived objects, we reinterpret four of the construction states as follows:

1. Unconstructed: Construction of the *non-virtual part* has not started yet. However, virtual bases may have been already constructed.
2. StartedConstructing: The construction of *non-virtual* base-class subobjects has started, but not the fields
6. DestructingBases: The fields have been completely destructed. *Non-virtual bases* are undergoing destruction.
7. Destructed: All fields and *non-virtual bases* have been destructed.

Then, the *lifetime* of a subobject σ can be defined as the set of execution states where its construction state is exactly Constructed.

Construction states are naturally ordered by chronology. We write $c < c'$ to say that state c occurs earlier than state c' in the enumeration above, and $\mathsf{S}(c)$ to denote the state immediately following c, if it exists.

Finally, fields also carry a construction state: one among Unconstructed, StartedConstructing, Constructed, StartedDestructing, and Destructed, with similar meaning as for subobjects. As an example of use, writing to a scalar field is possible only if it is in state Constructed, therefore preventing accesses to an unconstructed or already destructed field.

3.4 Operational semantics

The semantics of the core language is stated in terms of evolutions of a global state $\mathcal{G}$, which contains four partial maps:

- LocType: maps locations ℓ of complete objects to pairs (C, n) representing the types $C[n]$ of the complete objects.
- FieldValue: associates values to pairs (π, f) of a subobject π and a scalar field f.
- ConstrState: associates construction states to subobjects π.
- $\mathsf{ConstrState}^{\mathcal{F}}$: associates construction states to fields (π, f).

We write e.g. $\mathcal{G}.\mathsf{LocType}(\ell) = (C, n)$ to denote a lookup in a component of the state, and $\mathcal{G}[\mathsf{LocType}(\ell) \leftarrow (C, n)]$ to denote an update.

The main challenge in this semantics is to formalize the object construction and destruction protocol. (Other aspects of our language, such as accesses to fields, dynamic casts and virtual function dispatch, are semantically well understood from the work of Wasserrab *et al.* [18].) At a high level of abstraction, construction of a complete object corresponds to enumerating its subobjects according to a depth-first, left-to-right traversal of the construction tree depicted in Figure 1. The initializers encountered during this traversal are then executed to determine the arguments to the constructors, followed by invocations of the designated constructors (for bases and elements of structure array fields) or field assignments (for scalar fields). Construction states of subobjects and fields are updated along the way. Destruction is similar, but proceeds in the exact reverse order: depth-last, right-to-left traversal of the construction tree.

The description above strongly suggests a big-step operational semantics or, equivalently, a definitional interpreter, since the recursive structure of these semantics matches well the recursive nature of the construction tree. However, there are two reasons why we want a small-step transition semantics instead. First, statements, constructors and initializers may fail to terminate, and we would like to account both for terminating and diverging executions. Second and more importantly, we need our semantics to materialize the *context* in which a constructor or destructor executes, or in other terms the *continuation* of all pending constructor/destructor invocations which will be restarted when the current constructor/destructor finishes. Being able to reason on this explicit context/continuation is necessary in order to prove most of the high-level semantic properties of §4.

The operational semantics, therefore, is presented in small-step style as a transition relation $(\mathcal{P}, \mathcal{K}, \mathcal{G}) \to (\mathcal{P}', \mathcal{K}', \mathcal{G}')$ operating over triples of a control point $\mathcal{P}$, a continuation $\mathcal{K}$ and a global state $\mathcal{G}$. We distinguish 5 kinds of control points $\mathcal{P}$:

- $\mathsf{Codepoint}(\mathit{Stmt}_1, \mathit{Stmt}^*, \mathit{Env}, \mathit{Block}^*)$: about to execute statement Stmt_1 followed by the list of statements Stmt^*, under variable environment Env. Block^* is the list of all blocks enclosing the current statement, where a block is a pair (ℓ, Stmt^*) of a complete object ℓ to destruct at block exit and the remaining statements to execute after exiting from the block.
- $\mathsf{Constr}(\pi, \mathit{ItemKind}, \kappa, L, \mathit{Env})$: about to construct the list L of the bases or fields of the subobject π. Initializers are to be looked for using constructor κ, and they operate on the variable environment Env to pass arguments to their constructors. $\mathit{ItemKind}$ is one of $\mathsf{Bases}(\mathsf{DirectNonVirtual})$ or $\mathsf{Bases}(\mathsf{Virtual})$ or Fields, to request the construction of, respectively, direct non-virtual bases or virtual bases or fields of π.
- $\mathsf{ConstrArray}(\ell, \alpha, n, i, C, \mathit{ObjInit}^*, \mathit{Env})$: about to construct cells i to $n-1$ of type C, of the array α from the complete object ℓ, using the initializers $\mathit{ObjInit}^*$ to initialize the cells, and Env as variable environment to execute the initializers.
- $\mathsf{Destr}(\pi, \mathit{ItemKind}, L)$: about to destruct the list L of bases or fields of the subobject π.
- $\mathsf{DestrArray}(\ell, \alpha, i, C)$: about to destruct cells i down to 0 of type C, of the array α from the complete object ℓ.

We omit the grammar of continuations $\mathcal{K}$, which can be found in [12, chapter 9].

The operational semantics is composed of 56 inference rules defining the transition relation $(\mathcal{P}, \mathcal{K}, \mathcal{G}) \to (\mathcal{P}', \mathcal{K}', \mathcal{G}')$, for a total of about 900 lines of Coq definitions. We show some represen-

tative rules to give the general flavor, and refer the reader to the first author's thesis [12, chapter 9] for a full listing.

Field accesses Reading from a scalar field is modeled as follows:

$$\frac{\begin{array}{c}Env(var) = \pi \qquad \mathcal{G} \vdash \pi : C \qquad f = (fid, (\mathsf{Sc}, t)) \in \mathcal{F}(C) \\ \mathcal{G}.\mathsf{FieldValue}(\pi, f) = res \qquad Env' = Env[var' \leftarrow res]\end{array}}{\begin{array}{c}(\mathsf{Codepoint}(var' := var\text{->}_C f\ , L, Env, \mathcal{B}), \mathcal{K}, \mathcal{G}) \\ \rightarrow (\mathsf{Codepoint}(\mathsf{skip}\ , L, Env', \mathcal{B}), \mathcal{K}, \mathcal{G})\end{array}}$$

Likewise, for an assignment to a scalar field, we have:

$$\frac{\begin{array}{c}Env(var) = \pi \qquad \mathcal{G} \vdash \pi : C \qquad f = (fid, (\mathsf{Sc}, t)) \in \mathcal{F}(C) \\ \mathcal{G}.\mathsf{ConstrState}^{\mathcal{F}}(\pi, f) = \mathsf{Constructed} \\ Env(var') = res \qquad \mathcal{G}' = \mathcal{G}[\mathsf{FieldValue}(\pi, f) \leftarrow res]\end{array}}{\begin{array}{c}(\mathsf{Codepoint}(var\text{->}_C f := var'\ , L, Env, \mathcal{B}), \mathcal{K}, \mathcal{G}) \\ \rightarrow (\mathsf{Codepoint}(\mathsf{skip}\ , L, Env, \mathcal{B}), \mathcal{K}, \mathcal{G}')\end{array}}$$

Note the side condition preventing assignment to a scalar field that is not Constructed. For reads, this conditions is not necessary: Theorem 4 below shows that if a field has a value, then it is Constructed.

Dynamic types Virtual function calls and dynamic casts both involve the notion of *dynamic type* of a subobject, which the C++ standard defines as its most-derived object during the lifetime of the latter. However, the standard also permits virtual function calls and dynamic casts during the execution of the constructor body, or field initializers, or destructor body corresponding to the subobject. To unify these two cases, we introduce the notion of *generalized dynamic type*. We define the predicate $\mathsf{gDynType}(\ell, \alpha, i, \sigma, B, \sigma', \sigma'')$, meaning that the base-class subobject σ' of static type B is the generalized dynamic type of the subobject $(\ell, (\alpha, i, \sigma))$ with $\sigma = \sigma' @ \sigma''$.

$$\frac{\begin{array}{c}\mathcal{G}.\mathsf{LocType}(\ell) = (D, n) \qquad D[n] \dashv\langle\alpha\rangle^{\mathcal{A}}\!\rightarrow C[m] \dashv\langle(i, \sigma)\rangle^{\mathcal{C}\mathcal{I}}\!\rightarrow B \\ \mathcal{G}.\mathsf{ConstrState}(\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) = \mathsf{Constructed}\end{array}}{\mathcal{G} \vdash \mathsf{gDynType}(\ell, \alpha, i, \sigma, C, (\mathsf{Repeated}, C :: \epsilon), \sigma)}$$

$$\frac{\begin{array}{c}\mathcal{G}.\mathsf{LocType}(\ell) = (D, n) \\ D[n] \dashv\langle\alpha\rangle^{\mathcal{A}}\!\rightarrow C[m] \dashv\langle(i, \sigma_\circ)\rangle^{\mathcal{C}\mathcal{I}}\!\rightarrow C_\circ \\ \mathcal{G}.\mathsf{ConstrState}(\ell, (\alpha, i, \sigma_\circ)) = c \\ c = \mathsf{BasesConstructed} \vee c = \mathsf{StartedDestructing} \\ C_\circ \dashv\langle\sigma'\rangle^{\mathcal{I}}\!\rightarrow B \qquad \sigma = \sigma_\circ @ \sigma'\end{array}}{\mathcal{G} \vdash \mathsf{gDynType}(\ell, \alpha, i, \sigma, C_\circ, \sigma_\circ, \sigma')}$$

The semantics of virtual function calls is similar to that given by Wasserrab *et al.* [18], except that we use the generalized dynamic type instead of the most-derived object. If $C_\circ$ is the generalized dynamic type of subobject σ', the predicate $\mathsf{VFDispatch}(C_\circ, \sigma', f, B'', \sigma'')$ determines the subobject $\sigma'' : B''$ of $C_\circ$ containing the definition of virtual function f that must be invoked, following the same algorithm as in [18]:

1. Determine the static resolving subobject σ_f declaring f.
2. Choose the *final overrider* for the method. The final overrider is the inheritance subobject σ'' between $C_\circ$ and σ_f, nearest to $C_\circ$, and declaring f.

The transition rule for a virtual function call is, then,

$$\frac{\begin{array}{c}Env(var) = (\ell, (\alpha, i, \sigma)) \\ \mathcal{G} \vdash \mathsf{gDynType}(\ell, \alpha, i, \sigma, C_\circ, \sigma_\circ, \sigma') \\ \mathsf{VFDispatch}(C_\circ, \sigma', f, B'', \sigma'') \\ B''.f = f(varg_1, \ldots, varg_n)\{body\} \qquad \forall j, Env(var_j) = v_j \\ Env' = \varnothing[varg_1 \leftarrow v_1] \ldots [varg_n \leftarrow v_n][\mathsf{this} \leftarrow (\ell, (\alpha, i, \sigma_\circ @ \sigma''))]\end{array}}{\begin{array}{c}(\mathsf{Codepoint}(var\text{->}_B f(var_1 \ldots var_n), L, Env, \mathcal{B}), \mathcal{K}, \mathcal{G}) \\ \rightarrow (\mathsf{Codepoint}(body, \epsilon, Env', \epsilon), \mathsf{Kretcall}(var, L, Env, \mathcal{B}) :: \mathcal{K}, \mathcal{G})\end{array}}$$

Similarly, dynamic casts are handled as in Wasserrab *et al.* [18], using the generalized dynamic type instead of the most derived object. The transition rule is omitted for brevity, but can be found in [12, chapter 9].

Destruction We now give the flavor of the object construction and destruction protocol, starting with the simpler of the two, namely destruction. Destruction starts when the body of a block has reduced to skip or return:

$$\frac{(st = \mathsf{skip} \wedge L = \epsilon) \vee st = \mathsf{return} \qquad \mathcal{G}.\mathsf{LocType}(\ell) = (C, n)}{\begin{array}{c}(\mathsf{Codepoint}(st, L, Env, (\ell, L') :: \mathcal{B}), \mathcal{K}, \mathcal{G}) \\ \rightarrow (\mathsf{DestrArray}(\ell, L, n-1, C), \\ \mathsf{Kcontinue}(st, Env, L', \mathcal{B}) :: \mathcal{K}, \mathcal{G})\end{array}}$$

The DestrArray execution state requests the destruction of all elements of the structure array $C[n]$ at ℓ, starting with the last element $(n-1)$. Eventually, we reach a state where no more elements remain to be destructed, in which case we effectively exit from the block.

$$\frac{}{\begin{array}{c}(\mathsf{DestrArray}(\ell, \alpha, -1, C), \\ \mathsf{Kcontinue}(st, Env, L, \mathcal{B}) :: \mathcal{K}, \mathcal{G}) \\ \rightarrow (\mathsf{Codepoint}(st, L, Env, \mathcal{B}), \mathcal{K}, \mathcal{G})\end{array}}$$

When the destruction of a most-derived object (i.e. a structure array cell) is requested, we first enter the body of the associated destructor in a variable environment that binds this to the object.

$$\frac{\begin{array}{c}0 \leq i \qquad \sim C()\{stmt\} \\ \pi = (\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) \qquad Env = \varnothing[\mathsf{this} \leftarrow \pi] \\ \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{StartedDestructing}]\end{array}}{\begin{array}{c}(\mathsf{DestrArray}(\ell, \alpha, i, C), \mathcal{K}, \mathcal{G}) \\ \rightarrow (\mathsf{Codepoint}(stmt, \epsilon, Env, \epsilon), \\ \mathsf{Kdestr}(\pi) :: \mathsf{Kdestrcell}(\ell, \alpha, i, C) :: \mathcal{K}, \mathcal{G}')\end{array}}$$

When the destructor body returns, the fields of the subobject have to be destructed, in reverse declaration order.

$$\frac{\pi = (\ell, (\alpha, i, (h, l))) \qquad \mathsf{last}(l) = C \qquad L = \mathsf{rev}(\mathcal{F}(C))}{\begin{array}{c}(\mathsf{Codepoint}(\mathsf{return}, Stmt^*, Env, \epsilon), \mathsf{Kdestr}(\pi) :: \mathcal{K}, \mathcal{G}) \\ \rightarrow (\mathsf{Destr}(\pi, \mathsf{Fields}, L), \mathcal{K}, \mathcal{G})\end{array}}$$

Destructing a scalar field erases its value and changes its construction state to Destructed, before proceeding with the remaining fields.

$$\frac{\begin{array}{c}f = (fid, (\mathsf{Sc}, t)) \\ \mathcal{G}' = \mathcal{G}[\mathsf{FieldValue}(\pi, f) \leftarrow \bot][\mathsf{ConstrState}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{Destructed}]\end{array}}{\begin{array}{c}(\mathsf{Destr}(\pi, \mathsf{Fields}, f :: L), \mathcal{K}, \mathcal{G}) \\ \rightarrow (\mathsf{Destr}(\pi, \mathsf{Fields}, L), \mathcal{K}, \mathcal{G}')\end{array}}$$

Destructing a structure array field changes its construction state to StartedDestructing, then requests the destruction of the array, starting from its last cell, and remembering the remaining fields through Kdestrother in the continuation.

$$\frac{\begin{array}{c}\pi = (\ell, (\alpha, i, \sigma)) \\ f = (fid, (\mathsf{St}, (C, n))) \qquad \alpha' = \alpha +\!\!+ (i, \sigma, f) :: \epsilon \\ \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{StartedDestructing}]\end{array}}{\begin{array}{c}(\mathsf{Destr}(\pi, \mathsf{Fields}, f :: L), \mathcal{K}, \mathcal{G}) \\ \rightarrow (\mathsf{DestrArray}(\ell, \alpha', n-1, C), \\ \mathsf{Kdestrother}(\pi, \mathsf{Fields}, f, L) :: \mathcal{K}, \mathcal{G}')\end{array}}$$

Then, once all cells have been destructed, the field enters the Destructed state, and we proceed with the destruction of the remaining fields.

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{Destructed}]}{(\mathsf{DestrArray}(\ell', \alpha', -1, C), \mathsf{Kdestrother}(\pi, \mathsf{Fields}, f, L) :: \mathcal{K}, \mathcal{G}) \rightarrow (\mathsf{Destr}(\pi, \mathsf{Fields}, L), \mathcal{K}, \mathcal{G}')}$$

Eventually, all fields have been destructed. The subobject then changes its construction state to DestructingBases. At this point, no virtual function call nor dynamic casts may be used on this subobject. The destruction of the direct non-virtual bases starts, in reverse declaration order.

$$\frac{\pi = (\ell, (\alpha, i, (h, l))) \qquad \mathsf{last}(l) = C \qquad L = \mathsf{rev}(\mathcal{DNV}(C)) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{DestructingBases}]}{(\mathsf{Destr}(\pi, \mathsf{Fields}, \epsilon), \mathcal{K}, \mathcal{G}) \rightarrow (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), L), \mathcal{K}, \mathcal{G}')}$$

Destructing a (virtual or direct non-virtual) base B of π enters its destructor, remembering the other bases through Kdestrother.

$$\frac{\sim B()\{stmt\} \qquad \pi' = \mathsf{AddBase}(\pi, \beta, B) \qquad Env = \varnothing[\mathsf{this} \leftarrow \pi'] \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi') \leftarrow \mathsf{StartedDestructing}]}{(\mathsf{Destr}(\pi, \mathsf{Bases}(\beta), B :: L), \mathcal{K}, \mathcal{G}) \rightarrow (\mathsf{Codepoint}(stmt, \epsilon, Env, \epsilon), \mathsf{Kdestr}(\pi') :: \mathsf{Kdestrother}(\pi, \mathsf{Bases}(\beta), B, L) :: \mathcal{K}, \mathcal{G}')}$$

Eventually, we reach a state where all direct non-virtual bases of π have been destructed. There are two cases to consider, depending on the top of the continuation stack. In the first case, the continuation stack starts with a $\mathsf{Kdestrother}(\pi', \mathsf{Bases}(\beta), B, L)$, indicating that π is not a most-derived object. In this case, there is no need to destruct the virtual part of π (this will be done later by the enclosing most-derived object) and we are done with the subobject π: its construction state becomes Destructed, and we proceed with the destruction of the remaining bases of π'.

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{Destructed}]}{(\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), \epsilon), \mathsf{Kdestrother}(\pi', \mathsf{Bases}(\beta), B, L) :: \mathcal{K}, \mathcal{G}) \rightarrow (\mathsf{Destr}(\pi', \mathsf{Bases}(\beta), L), \mathcal{K}, \mathcal{G}')}$$

The second case is when the continuation stack starts with a Kdestrcell. Then, π must be a most-derived object, and its direct and indirect virtual bases need to be destructed. According to the C++ standard, virtual bases must be destructed in *reverse inheritance graph order*. Consider the following recursive function:

$$\begin{aligned} \mathsf{list}(\{\mathsf{Repeated}, \mathsf{Shared}\} \times \mathcal{C}) &\overset{\mathcal{VO}}{\rightarrow} \mathsf{list}(\mathcal{C}) \\ \mathcal{VO}(\epsilon) &= \epsilon \\ \mathcal{VO}((\mathsf{Repeated}, B) :: q) &= \mathcal{VO}(\mathcal{D}(B)) +\!\!+' \mathcal{VO}(q) \\ \mathcal{VO}((\mathsf{Shared}, B) :: q) &= \mathcal{VO}(\mathcal{D}(B)) +\!\!+' (B :: \mathcal{VO}(q)) \end{aligned}$$

where $\mathcal{D}(B)$ is the list of direct bases of B, in declaration order, tagged with Repeated(for non-virtual bases) or Shared(for virtual bases), and $+\!\!+'$ is list concatenation with elimination of duplicates ($l_1 +\!\!+' l_2 =_{\text{def}} l_1 +\!\!+ (l_2 \setminus l_1)$). It is easy to see that the list $L = \mathcal{VO}(\mathcal{D}(C))$ contains all the direct and indirect virtual bases of C, exactly once each, and contains no other classes. Moreover, the order in which virtual bases appear in list L is consistent with the inheritance graph order. In particular, if A and B are virtual bases of C such that A is a virtual base of B, then A appears before B in L.

$$\frac{L' = \mathsf{rev}(\mathcal{VO}(\mathcal{D}(C)))}{(\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), \epsilon), \mathsf{Kdestrcell}(\ell, \alpha, i, C) :: \mathcal{K}, \mathcal{G}) \rightarrow (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), L'), \mathcal{K}, \mathcal{G})}$$

Finally, when all virtual bases have been destructed, the subobject under consideration (necessarily a most-derived object) becomes Destructed, and we proceed with the destruction of the preceding structure array cells.

$$\frac{\pi = (\ell, (\alpha, i, (h, l))) \qquad \mathsf{last}(l) = C \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{Destructed}]}{(\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \epsilon), \mathcal{K}, \mathcal{G}) \rightarrow (\mathsf{DestrArray}(\ell, \alpha, i-1, C), \mathcal{K}, \mathcal{G}')}$$

Construction To execute a block statement, we allocate memory for the structure array declared in the block, then start the construction protocol by requesting the construction of the first element of this array:

$$\frac{\ell \notin \mathsf{dom}(\mathcal{G}.\mathsf{LocType}) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{LocType}(\ell) \leftarrow (C, n)] \qquad Env' = Env[c \leftarrow \mathsf{Ptr}(\ell, \epsilon, (\mathsf{Repeated}, C :: \epsilon))]}{(\mathsf{Codepoint}(\{C\ c[n] = \{\iota\}; st\}, St, Env, Bl), \mathcal{K}, \mathcal{G}) \rightarrow (\mathsf{ConstrArray}(\ell, \epsilon, n, 0, C, \iota, Env'), \mathsf{Kcontinue}(st, Env', St, Bl) :: \mathcal{K}, \mathcal{G}')}$$

The construction protocol implements the necessary traversal of all subobjects of c using the same "small-stepping" style, based on continuations, that was illustrated above in the case of destruction. There are two main differences. First, the construction order is the exact opposite of the destruction order: instead of enumerating array cells by decreasing indices, fields and non-virtual bases in reverse declaration order and virtual bases in reverse $\mathcal{VO}$ order, we enumerate array cells by increasing indices, fields and non-virtual bases in declaration order and virtual bases in $\mathcal{VO}$ order. Second and more importantly, we need to execute the initializers that compute initial values for scalar fields and the arguments to be passed to constructors, adding a number of transition rules. We refer the reader to [12, chapter 9] for full details.

4. Properties of construction and destruction

In this section, we state (and sometimes outline the proofs of) several semantic properties of interest for construction and destruction. Some are technical consequences of the definition of our semantics, but others capture higher-level properties that C++ programmers often rely on, such as the RAII principle.

4.1 Run-time invariant

Our transition rules and grammars for execution states are lax in that they put very few constraints on the general shapes of states. However, in transition sequences starting from the initial state, the reachable states are a small subset of all possible states and enjoy many low-level properties that are essential to prove the high-level theorems in this section. We gathered about 20 such properties in one invariant INV and proved that this invariant is satisfied by the initial state and preserved by the transition rules. This is the largest part of the Coq mechanization, totaling about 15000 lines of Coq and taking about 2 hours to re-check the proof.

A detailed explanation of the invariant is given in [12, section 10.1]. Here, we just show the two most interesting consequences of the invariant, which relate the construction states of certain subobjects.

Let p, p' be two subobjects of the same complete object. We say that p is a *direct subobject* of p' if, either, (1) p is a direct non-virtual base of p'; (2) or p' is a most-derived object and p is a virtual base of p'; (3) or p is a field subobject of p' (that is, p is a cell of a structure array field of p').

Lemma 1 (Vertical relations on construction states)**.** *Assume that p is a direct subobject of p'. The construction states of p and p' in any execution state satisfying INV are related as follows:*

If p' is...	*Then p is...*
Unconstructed	*Unconstructed*
StartedConstructing	*Unconstructed if p is field subobject of p'*
BasesConstructed	*Constructed if p is a base subobject of p'*
Constructed	*Constructed*
StartedDestructing	*Constructed if p is a base subobject of p'*
DestructingBases	*Destructed if p is a field subobject of p'*
Destructed	*Destructed*

Let p' be a subobject of static type C, and p_1, p_2 be two direct subobjects of p'. We say that p_1 *occurs before* p_2 if, either, (1) p' is a most-derived object and p_1, p_2 are two virtual bases of p' in inheritance graph order; (2) p_1 and p_2 are two direct non-virtual bases of p in declaration order; (3) p_1 and p_2 are two cells of the same array field, in the order of their indexes within the array; (4) p_1 and p_2 are two cells of two different fields in declaration order; (5) p' is a most-derived object and p_1 is a virtual base, and p_2 is a direct non-virtual base of p' or a cell of an array field; (6) p_1 is a direct non-virtual base of p' and p_2 is a cell of an array field.

Lemma 2 (Horizontal relations on construction states)**.** *Assume that p_1 occurs before p_2. The construction states of p_1 and p_2 in any execution state satisfying INV are related as follows:*

If p_1 is...	*Then p_2 is...*
Unconstructed *StartedConstructing* *BasesConstructed*	*Unconstructed*
Constructed	*in an arbitrary state*
StartedDestructing *DestructingBases* *Destructed*	*Destructed*

In the remainder of this section, we only consider execution states that can be reached from the initial state and therefore satisfy the invariant INV.

4.2 Progress

To check that our transition rules make sense and that no rule is missing, we prove a "progress" property of construction and destruction: once construction of a complete object starts, it always eventually reaches a point where the object is fully constructed, without getting stuck in the middle; and likewise for destruction. This result is false in general: since constructors, initializers and destructors can perform arbitrary computation, they can get stuck on e.g. an attempt to assign an unconstructed scalar field. However, we can prove the expected result if we restrict ourselves to *nearly trivial* constructors and *trivial destructors*.

We say that a class C has a nearly trivial constructor if (1) C has a default constructor with no arguments and `return` as its body; (2) initializers for bases and fields just call the default constructors, without any other computations; (3) scalar fields are initialized with constants; (4) for each structure array field f of C, if f has type B, then B has a nearly trivial constructor. This notion extends the concept of *trivial constructor* from the C++ standard, e.g. by allowing virtual bases and virtual functions.

Theorem 3 (Construction progress)**.** *Let $(\mathcal{P}, \mathcal{K}, \mathcal{G})$ be an execution state where $\mathcal{P} = \mathsf{Codepoint}(\{C\ c[n]; st\} = \iota, \ldots)$, i.e. we are about to execute a block statement. Assume that C is a class having a nearly trivial constructor and that the initializer list ι calls the default constructor for every array cell. Then, there exists a state $(\mathcal{P}', \mathcal{K}', \mathcal{G}')$ such that*

- $(\mathcal{P}, \mathcal{K}, \mathcal{G}) \xrightarrow{*} (\mathcal{P}', \mathcal{K}', \mathcal{G}')$
- *$\mathcal{P}' = \mathsf{Codepoint}(st, \ldots)$, i.e. construction has terminated and the block body is about to be executed;*
- *in state $\mathcal{G}'$, all cells of the array c are in the Constructed state.*

A similar theorem holds for destruction. Here, we use unchanged the notion of *trivial destructor* from the C++ standard: a class C has a trivial destructor if (1) its destructor is just `return`; (2) all virtual bases and direct non-virtual bases of C have trivial destructors; (3) for each structure array field f of C, if f has type B, then B has a trivial destructor.

4.3 Safety of field accesses and virtual function calls

The rule for reading the contents of a scalar field puts no precondition on the initialization state of the field. We can, however, show that this rule gets stuck whenever the field is not in the Constructed state.

Theorem 4. *A scalar field has a well-defined value only if it is Constructed.*

Proof. Follows from INV and the fact that setting the value of a scalar field only occurs in two cases: either on an ordinary scalar field, which is forbidden if the field is not Constructed, or on scalar field initialization, which turns the field construction state to Constructed. Moreover, our semantics erases the value of the scalar field when destructing it. □

Like C++ itself, our semantics allows fields to have no initializers and therefore remain without a defined value after construction. However, we also proved that if we remove the transition rule allowing fields without initializers to proceed, a scalar field has a well-defined value if and only if it is Constructed.

Virtual functions have no initialization state *per se*. However, the C++ semantics for virtual function calls provides a very useful guarantee concerning the construction state of the `this` subobject:

Theorem 5. *Whenever a virtual function is called, the subobject bound to its `this` parameter is in state BasesConstructed or Constructed or StartedDestructing, and all its base-class subobjects are in state Constructed.*

Proof. Consider a call to a virtual function f on a subobject (ℓ, p). Since this call does not go wrong, the generalized dynamic type p_o of this subobject is defined. By definition of generalized dynamic types, (ℓ, p_o) is in state BasesConstructed or Constructed or StartedDestructing. Invariant INV then guarantees that all base-class subobjects of (ℓ, p_o) are Constructed (from Lemma 1). The final overrider for function f being either (ℓ, p_o) or one of its base-class subobjects, the result follows. □

In other words, a virtual function can always safely assume that bases have been properly constructed. It cannot, however, assume that fields of its defining class are in the constructed state, since initializers for these fields can call the virtual function. As discussed in §2 and by Qi and Myers [11], Java does not provide a similar guarantee; through inheritance and method overriding, it is possible for a method to be invoked before its super classes have completed their initialization.

4.4 Evolution of construction states

Lemma 6. *If $s \to s'$ is a transition step of our small-step operational semantics, and if the construction state of the subobject (ℓ, p) is c in s and $c' \neq c$ in s', then $c' = \mathsf{S}(c)$ and any other subobject $(\ell', p') \neq (\ell, p)$ keeps its construction state unchanged.*

Proof. By case analysis on the transition rules, with the help of invariant INV. □

Corollary 7. *Any subobject is never constructed or destructed more than once.*

In particular, any virtual base subobject is constructed or destructed at most once, despite being potentially reachable through several inheritance paths. Also, if an object goes from one construction state to another, then it must go through all construction states in between:

Theorem 8 (Intermediate values theorem)**.** *If $s \xrightarrow{*} s'$, then, for any subobject (ℓ, p) and for any construction state c such that:*

$$\mathsf{ConstrState}_s(\ell, p) \leq c < \mathsf{S}(c) \leq \mathsf{ConstrState}_{s'}(\ell, p)$$

there exist "changing states" s_1, s_2 such that $s \xrightarrow{*} s_1 \to s_2 \xrightarrow{*} s'$ and $\mathsf{ConstrState}_{s_1}(\ell, p) = c$ and $\mathsf{ConstrState}_{s_2}(\ell, p) = \mathsf{S}(c)$.

(Here, for a state $s = (\mathcal{P}, \mathcal{K}, \mathcal{G})$, we write $\mathsf{ConstrState}_s(\pi)$ for $\mathcal{G}.\mathsf{ConstrState}(\pi)$.)

4.5 Object lifetimes

An important design principle of C++ is that destruction is performed in the exact reverse order of construction. We now formalize and prove this property, along with more general properties about the relative lifetimes of two subobjects.

Two subobjects of the same complete object Let ℓ be a complete object of type $C[n]$.

Theorem 9. *If p_1 and p_2 are two subobjects of the complete object ℓ, either the lifetime of p_1 is included in that of p_2, or the lifetime of p_2 is included in that of p_1.*

To prove this theorem, we need to characterize the construction order between subobjects. We write $p_1 \preceq_{C[n]} p_2$ to say that p_1 occurs before p_2 in the depth-first, left-to-right traversal of the construction tree in Figure 1. (See [12, chapter 10] for a formal definition in terms of the "direct subobject" and "occurs before" relations of §4.1.) The two crucial properties of this construction order are the following:

Lemma 10 (The construction order is total)**.** *If $C[n] \multimap\!\langle p_1 \rangle\!\rightarrowtail B_1$ and $C[n] \multimap\!\langle p_2 \rangle\!\rightarrowtail B_2$, then either $p_1 \preceq_{C[n]} p_2$ or $p_2 \preceq_{C[n]} p_1$.*

Lemma 11 (Construction order and lifetimes)**.** *For any reachable execution state s and any complete object ℓ of type $C[n]$, if $p_1 \preceq_{C[n]} p_2$ and $\mathsf{ConstrState}_s(\ell, p_2) = \mathsf{Constructed}$, then $\mathsf{ConstrState}_s(\ell, p_1) = \mathsf{Constructed}$. In other words, if $p_1 \preceq_{C[n]} p_2$, then the lifetime of p_2 is included in the lifetime of p_1.*

Theorem 9 then follows directly from the two lemmas above. Using Lemma 11 and the definition of $\preceq_{C[n]}$, we have the following stronger special case:

Theorem 12. *If p_2 is a subobject of p_1, then the lifetime of p_1 is included in that of p_2.*

As a corollary of Theorem 9, it follows that if p_1 is constructed before p_2, then p_2 is destructed before p_1.

Theorem 13 (Destruction in reverse order of construction)**.** *Let ℓ be a complete object of type $C[n]$ and p_1, p_2 be two subobjects of ℓ. Consider the reduction sequence below where p_1 is constructed before p_2, then later p_1 is destructed:*

$$\underset{(\neg c_1)}{s_0} \to \underset{(c_1)}{s_1} \xrightarrow{*} \underset{(\neg c_2)}{s_2} \to \underset{(c_2)}{s_3} \xrightarrow{*} \underset{(c_1)}{s_4} \to \underset{(\neg c_1)}{s_5}$$

(We write c_i below a state s to denote that p_i is $\mathsf{Constructed}$ in state s, and $\neg c_i$ to mean that p_i is not $\mathsf{Constructed}$.) Then, there exist intermediate states s'_3, s'_4 such that p_2 stops being $\mathsf{Constructed}$ between these two states:

$$\underset{(\neg c_1)}{s_0} \to \underset{(c_1)}{s_1} \xrightarrow{*} \underset{(\neg c_2)}{s_2} \to \underset{(c_2)}{s_3} \xrightarrow{*} \underset{(c_2)}{s'_3} \to \underset{(\neg c_2)}{s'_4} \xrightarrow{*} \underset{(c_1)}{s_4} \to \underset{(\neg c_1)}{s_5}$$

Proof. At state s_2, p_1 is $\mathsf{Constructed}$ but p_2 is not. Hence, the lifetime of p_1 cannot be included in that of p_2. By theorem 9, the lifetime of p_2 is therefore included in that of p_1. Since p_1 is no longer $\mathsf{Constructed}$ at state s_5, so is p_2 at state s_5. Since at most one subobject changes construction state during a given transition (Lemma 6), $p_1 \neq p_2$ and p_2 is no longer $\mathsf{Constructed}$ at state s_4. The result then follows from the the intermediate values theorem (Theorem 8). □

Subobjects of different complete objects In the full C++ language, including dynamic allocation, the lifetimes of two subobjects of different complete objects are, in general, unrelated: `new` and `delete` operations can be interleaved arbitrarily. In our core language, complete objects can only be created by block statements and therefore follow a stack discipline.

Theorem 14. *The lifetimes of two subobjects (ℓ_1, p_1) and (ℓ_2, p_2) of different complete objects $\ell_1 \neq \ell_2$ are either disjoint or included in one another.*

This result follows from Theorem 9 and the stronger property below, which shows that throughout the execution of a block, the construction states of subobjects of already-allocated complete objects do not change:

Lemma 15. *Let $s_1 \to s_2 \xrightarrow{*} s_3 \to s_4$ be the execution of a block: the transition $s_1 \to s_2$ enters the block and allocates a fresh complete object ℓ; the transition $s_3 \to s_4$ exits this block. For all complete objects ℓ' already allocated in state s_1, the allocation states of its subobjects (ℓ', p') are identical in s_1 and s_4.*

4.6 RAII: Resource Acquisition Is Initialization

RAII is a programming discipline where precious program resources (such as file descriptors) are systematically encapsulated in classes, all acquisitions of such resources are performed within constructors of the corresponding class, and all releases of such resources are performed within destructors of the corresponding class. We cannot prove a general result guaranteeing the proper encapsulation of resources in classes: this is a matter of program verification. We can, however, prove that in a terminating program every construction of a subobject is correctly matched by a destruction.

Theorem 16 (RAII)**.** *Consider a partial program execution $s_{init} \xrightarrow{*} s_1 \to s_2$ where the transition $s_1 \to s_2$ marks the end of a block statement that allocated the complete object ℓ. Then, between the initial state s_{init} and s_1, the following events occurred, in this order:*

- *Every subobject of ℓ was constructed exactly once.*
- *Every subobject of ℓ was destructed exactly once, in reverse order of construction.*

Proof. The invariant INV implies that, at state s_1, all cells of the structure array ℓ are $\mathsf{Destructed}$, and so are all of their subobjects. Consider a subobject (ℓ, p). Its state is $\mathsf{Unconstructed}$ in the initial state and $\mathsf{Destructed}$ in s_1. Therefore, by the intermediate value theorem (Theorem 8), it must have entered state $\mathsf{Constructed}$ at some point, then left state $\mathsf{Constructed}$ at some later point, meaning that (ℓ, p) has been initialized once, then destructed once. The claim on construction and destruction order follows from Theorem 13. □

4.7 Generalized dynamic types

We finish this section with interesting properties of generalized dynamic types, which were introduced in §3.4 to give semantics to virtual function calls and dynamic casts.

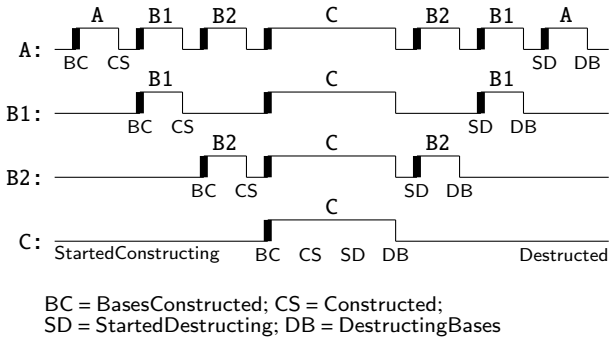

Figure 2. Evolution of the dynamic types of an instance of `C` and of its subobjects in the example of §4.7. Dynamic types are undefined at points where the "signal" is low, and defined and equal to the indicated type when the "signal" is high. Thick vertical transitions denote points where a compiled implementation must update pointers to v-tables.

Theorem 17. *The generalized dynamic type of a subobject, if defined, is unique.*

Proof. The result is trivial if the most-derived object is Constructed, since it is then equal to the generalized dynamic type. Otherwise, we observe that the most-derived object can have at most one base class subobject in state BasesConstructed or StartedDestructing (as a consequence of invariant INV), and conclude. □

A subtle aspect of generalized dynamic types is that they do not *continuously* exist: as the construction or destruction of a most-derived object proceeds, the generalized dynamic type of one of its base-class subobjects alternates between defined and undefined. Consider the following program:

```
struct A             {virtual void f ();};
struct B1: virtual A {};
struct B2: virtual A {virtual void f ();};
struct C: B1, B2 {}
```

Figure 2 shows the evolution of the dynamic type of an instance of `C` and of its subobjects while this instance undergoes construction then destruction. For example, during the construction of base `B2`, the subobject `B1` is already Constructed, but its generalized dynamic type is undefined (the constructor of `B1` has returned but `C` is not yet constructed), and calling `f` on `B1` has undefined behavior.

Theorem 18. *Let $(\ell, (\alpha, i, \sigma))$ be a subobject and $(\ell, (\alpha, i, \sigma_\circ))$ be its most derived object (i.e. $\sigma_\circ = (\mathsf{Repeated}, C :: \epsilon)$). Consider a transition $s \to s'$ where the construction state of $(\ell, (\alpha, i, \sigma))$ changes. Then, the generalized dynamic types for all subobjects in the program evolve as shown in Table 1.*

In a compiled implementation, dynamic types are materialized as extra fields in the in-memory representations of objects, these fields containing (in general) pointers to v-tables. When the generalized dynamic type of a subobject is undefined, its dynamic type field can contain any value. However, when the generalized dynamic type is defined, the dynamic type field must contain a pointer to the corresponding v-table. Theorem 18, therefore, pinpoints exactly the program points where the compiled code must update dynamic type fields: when all bases of a subobject are constructed, and just before the construction of fields begins, dynamic type fields must be updated for the subobject in question and all of its bases; likewise, when the subobject undergoes destruction, at the point where the destructor is entered.

5. Impact on the C++ language specification

The development of the formal semantics reported in this paper uncovered several issues with the C++03 standard. They were reported to the ISO C++ committee. Some of them were fixed in time for the C++11 standards; others will be addressed in future revisions.

Virtual functions calls during object construction and destruction The formulation of ISO C++03 [6] of the behavior of the abstract machine when a virtual function is called during construction (or destruction) was unclear and did not clearly support the original intent (correctly modeled in this paper). This was reported to the ISO C++ committee as CWG issue number 1202. The formulation was clarified in time for adoption in C++11.

Conflicting description of end of object lifetime Although the language formally defined in this paper does not allow explicit management of object lifetime, coming up with a simple, coherent, unified, and faithful description of object lifetime led us to discover conflicts and unintended semantics in the ISO C++ documents (both C++03 and C++11.) The C++11 standard [7] has a conflicting description of the effect of calling a destructor. Two paragraphs (3.8p1 and 3.8p4) claim that an object's lifetime ends when a call to a non-trivial destructor occurs or its storage is reused or released. However, another paragraph (14.2p4) claims that once a destructor is invoked (its triviality notwithstanding), "the object no longer exists". This issue will be resolved after C++11 is published. We expect that for consistency, the resolution will not consider "triviality" of the destructor to determine when an object's lifetime ends.

Effect of calling destructor for builtin types C++ allows an expression of the form `p->~T()` where `T` is a non-class type name and `p` is an expression that points to a `T` object. This form was introduced to support function templates explicitly managing object lifetime without forcing the author to distinguish between builtin types and class types. It was also intended to bring uniformity. However, the formulation of the behavior of the abstract machine *appears* to indicate that the following program fragment has a well-defined meaning, for any scalar type `T`.

```
T f(T x) {
  T t = x;
  t.~T();
  return t;
}
```

This is in a clear conflict with the case where `T` is a class type. Indeed, for every class type `T`, that function leads to an undefined behavior, as it attempts to destroy the local variable `t` twice: once explicitly through the call to the destructor and a second time implicitly when the function exits. Finally, it appears that the program is ill-formed (and a diagnostic is required) when `T` is an array type.

Lifetime of array objects The C++ standards explicitly indicate that the lifetime of an array object starts as soon as storage for it has been allocated, regardless of when the lifetime of its elements starts. In general, a complete object's lifetime starts after all its subobjects have been constructed. This irregularity appears to be a hand-over from C in its formulation. It will be reconsidered after publication of C++11 along with a more unified treatment of the lifetime of objects of builtin types.

6. Application to verified compilation

To strengthen confidence in the semantics presented here and stress its usability, the first author developed and proved correct a simple compiler that translates the core language presented in this paper

When the subobject	goes from	to	then the gen. dynamic type of	goes from	to
$(\ell, (\alpha, i, \sigma))$	Unconstructed	StartedConstructing	$(\ell, (\alpha, i, \sigma'))$	Undef.	Undef.
$(\ell, (\alpha, i, \sigma))$	StartedConstructing	BasesConstructed	$(\ell, (\alpha, i, \sigma@\sigma''))$	Undef.	σ
			$(\ell, (\alpha, i, \sigma'))$ not a base of σ	Undef.	Undef.
$(\ell, (\alpha, i, \sigma))$ with $\sigma \neq \sigma_\circ$	BasesConstructed	Constructed	$(\ell, (\alpha, i, \sigma@\sigma''))$	σ	Undef.
			$(\ell, (\alpha, i, \sigma'))$ not a base of σ	Undef.	Undef.
$(\ell, (\alpha, i, \sigma_\circ))$	BasesConstructed	Constructed	$(\ell, (\alpha, i, \sigma'))$	$\sigma_\circ$	$\sigma_\circ$
$(\ell, (\alpha, i, \sigma_\circ))$	Constructed	StartedDestructing	$(\ell, (\alpha, i, \sigma'))$	$\sigma_\circ$	$\sigma_\circ$
$(\ell, (\alpha, i, \sigma))$ with $\sigma \neq \sigma_\circ$	Constructed	StartedDestructing	$(\ell, (\alpha, i, \sigma@\sigma''))$	Undef.	σ
			$(\ell, (\alpha, i, \sigma'))$ not a base of σ	Undef.	Undef.
$(\ell, \alpha, i, \sigma)$	StartedDestructing	DestructingBases	$(\ell, (\alpha, i, \sigma@\sigma''))$	σ	Undef.
			$(\ell, (\alpha, i, \sigma'))$ not a base of σ	Undef.	Undef.
$(\ell, (\alpha, i, \sigma))$	DestructingBases	Destructed	$(\ell, (\alpha, i, \sigma'))$	Undef.	Undef.
$(\ell, (\alpha, i, \sigma))$	Any	Any	$(\ell', (\alpha', i', \sigma'))$ with $(\ell, \alpha, i) \neq (\ell', \alpha', i')$	Does not change	

Table 1. Evolutions of generalized dynamic types when the state of a subobject $(\ell, (\alpha, i, \sigma))$, of most-derived object $\sigma_\circ$, changes.

to a simple, non-object-oriented intermediate language similar to CompCert's Cminor [8]. This verification is described in [12, chapter 11]. The compiler proceeds in two passes, using as intermediate language the CoreC++ language of Wasserrab *et al.* [18] extended with a `setDynType` operation that explicitly modifies the dynamic type of an object.

The first translation pass is broadly similar to that outlined in Wasserrab's thesis [17], turning constructors, initializers and destructors into non-virtual function calls. However, `setDynType` operations are inserted at the program points characterized by Theorem 18 to implement the proper semantics for virtual function calls and dynamic casts. Following a popular optimization, two versions of every constructor are generated, one for the "most derived" case, the other for the "inheritance subobject" case. The second translation pass extends our earlier work on object layout [13].

The proofs of semantic preservation are standard arguments by forward simulation diagrams [8, §3.7]. The proofs are rather big (about 8000 lines of Coq for each pass) because there are many cases to consider, along with complex invariants relating execution states, but present no major conceptual difficulties.

7. Related work

Formal semantics for C++ Norrish [10] describes a formal semantics for C++ that was mechanized in HOL. His semantics describes a much larger subset of C++ than ours, including in particular expressions with side effects and partially-specified evaluation order, exceptions, free store (`new` and `delete`), and temporary objects. Construction and destruction are modeled by "dynamic translation": the reduction rule for the construction of an object produces "on the fly" a statement containing the necessary invocations of initializers and constructors for fields and bases; this statement is, then, reduced normally. As a way to state the semantics, Norrish's approach is arguably simpler than our "small-stepping" of the construction/destruction protocol. However, we suspect that Norrish's approach would make it more difficult to prove meta-properties about construction states and lifetimes, in the style of §4. Additionally, we suspect that Norrish's semantics does not correctly capture the interaction between construction, destruction, and virtual function calls. Consider:

```
struct B {
  B* b;
  virtual int f() { return 18; }
  B() : b(this) { }
};
struct D : B {
  virtual int f() { return 42; }
};
int main () { D d; return d.b->f(); };
```

In `B`'s constructor, the initialization `b(this)` saves in `b` a pointer whose dynamic type is `B` (correctly, at that time). However, after completion of `D`'s constructor, the dynamic type of `*b` is not updated to `D`, causing the wrong `f` virtual function to be dispatched in `main`.

The CoreC++ semantics of Wasserrab, Nipkow, Snelting and Tip [18] is a major starting point for our work. It does not cover object construction and destruction. Wasserrab's Ph.D. thesis [17] describes this semantics and its Isabelle/HOL formalization in greater details, as well as a static, unverified translation of construction and destruction into non-virtual function calls. This translation fails to correctly implement C++'s semantics for virtual function calls: in the translated program, the dynamic type of a subobject is always its most derived object, regardless of construction state.

We are not aware of any other formal semantics for C++ that addresses construction and destruction.

Type systems for safe object initialization Several research projects develop type systems for object-oriented languages that enforce safety guarantees about object initialization, such as the fact that well-typed programs never read fields of an uninitialized object. Fähndrich and Xia [3] introduce a type system to remove useless field initializations to `null` while still ensuring safe object initialization. Qi and Myers [11] introduce a more general type system to precisely and statically determine, at each program point, which fields may be read or not. Hubert *et al.* [5] formalize a type system for safe Java object initialization using the Coq proof assistant. Their type system shares with our operational semantics the use of construction states for objects, but in their case, construction states are lifted to the type level and maintained at compile-time. This enables their system to statically check contracts over methods that constrain the construction states of some of the arguments, for instance.

Our work makes no attempt at providing static typing safety properties beyond the few offered by C++. Instead, we aim at a precise description of the dynamic semantics of construction and destruction in the presence of multiple inheritance. The type systems mentioned above are based on the Java and C# single-inheritance object models. Moreover, they only deal with object initialization, without object destruction or finalization. Indeed, in Java, object finalization is weakly specified: although the Java language specification [4] requires objects to be finalized, it does

not describe more precisely when object finalization should occur. C# offers a destruction mechanism, namely object disposal, but it must be called explicitly by the programmer.

8. Extending the semantics

Our semantics presents the core features of C++ object construction and destruction as faithfully as possible towards the Standard. However, it can be extended in a number of directions.

Manual memory management We anticipate no difficulties with supporting free store objects, i.e. the `new` and `delete` operators. Only slight modifications to the operational semantics appear necessary. Of course, Theorem 14 would be invalidated, since objects no longer follow a stack discipline. However, it appears very difficult to formalize more general manual memory management, allowing for instance explicit destructor calls and the use of placement operators such as "`new(p) C`" to construct an object at a given memory location.

Temporary objects of class types Expression evaluation, passing arguments by value to functions, or returning results by value entail construction and destruction of *temporary objects*. While the lifetime of these objects are well-defined by the C++ standard (and they follow a first-constructed-last-destroyed discipline), their storage durations are not specified. In fact, we found inconsistencies between the definitions of storage duration as specified by the C++11 standards and the C99 standards. Accounting for temporary objects beyond scalar values returned by functions or passed as arguments will require nontrivial extensions to our semantics.

Copy constructor elision More challenging is to give semantics to functions that return values of class types. They are objects constructed in the callee but destroyed in the caller at the end of the full expression containing the call. For decades, C++ has allowed elision of copy constructors (even if they have observable behavior) that would normally copy a local object to the (temporary) return value, provided certain non-aliasing conditions (easy to check) are met. These program transformations are necessary in practice to obtain good performance but are not semantics-preserving in the traditional sense. The same caveat applies for the C++11 notion of "move constructors".

Exceptions It is easy to capture a key aspect of C++ exceptions in our semantics: block-scoped objects are properly destroyed when an exception is thrown. Exception objects are temporary objects with lifetime dynamically controlled by exception handlers acting much like function invocations with the exception object as argument. C++ allows catching exceptions by value, so our observations for function call argument by value apply here. Furthermore, the semantics of constructors need to be altered to invoke destructors for completely constructed subobjects in presence of exception. Similarly, the semantics for destructors should be altered to include program abortion if a subobject destructor raises an exception. Finally, interaction between construction of dynamic objects and exceptions (both of which reflect related design decisions) should be investigated. These features are at the basis of generalized RAII techniques for dynamically controlled resources; popular examples include smart pointers such as `unique_ptr` and `smart_ptr`.

9. Concluding remarks

We hope that this work sheds light on the precise semantics of construction and destruction in C++ and what they actually guarantee to the working programmer. Several features remain to be addressed, but the subset that was formalized is already quite realistic and similar to recommended subsets for critical embedded systems [9]. The semantics implements a pattern for "small-stepping" a tree traversal which could perhaps be generalized and abstracted over. Finally, it would be interesting to exploit this semantics in the context of type systems and other static analyses that verify stronger safety properties about initialization.

References

[1] The Coq proof assistant, 1999–2012. URL `http://coq.inria.fr`.

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

[3] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *22nd conf. on Object-Oriented Programming Systems and Applications (OOPSLA'07)*, pages 337–350. ACM, 2007.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition edition, 2005.

[5] L. Hubert, T. Jensen, V. Monfort, and D. Pichardie. Enforcing secure object initialization in Java. In *Computer Security – ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2010.

[6] *International Standard ISO/IEC 14882:2003. Programming Languages — C++*. International Organization for Standards, 2003.

[7] *International Standard ISO/IEC 14882:2011. Programming Languages — C++*. International Organization for Standards, 2011.

[8] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[9] Lockheed Martin. Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program, 2005. URL `http://www.research.att.com/~bs/JSF-AV-rules.pdf`.

[10] M. Norrish. A formal semantics for C++. Technical report, NICTA, 2008.

[11] X. Qi and A. C. Myers. Masked types for sound object initialization. In *36th symp. Principles of Programming Languages (POPL'09)*, pages 53–65. ACM, 2009.

[12] T. Ramananandro. *Mechanized Formal Semantics and Verified Compilation for C++ Objects*. PhD thesis, Université Paris Diderot, Jan. 2012.

[13] T. Ramananandro, G. Dos Reis, and X. Leroy. Formal verification of object layout for C++ multiple inheritance. In *38th symp. Principles of Programming Languages (POPL'11)*, pages 67–80. ACM, 2011.

[14] J. G. Rossie and D. P. Friedman. An algebraic semantics of subobjects. In *10th conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'95)*, pages 187–199. ACM, 1995.

[15] B. Stroustrup. Classes: An abstract data type facility for the C language. *SIGPLAN Not.*, 17:42–51, January 1982.

[16] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[17] D. Wasserrab. *From Formal Semantics to Verified Slicing – A Modular Framework with Applications in Language Based Security*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, Oct. 2010.

[18] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *21st conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 345–362. ACM, 2006.