A formal operational semantics for C++ object construction and destruction

Tahina Ramananandro
http://gallium.inria.fr/~tramanan/cpp

July 13, 2011

Notations and conventions

We adopt the usual Coq list notations:

- $\mathsf{list}(A)$ is the set of all lists having their elements in A
- nil is the empty list
- $a :: q$ is the list starting with an element a and continuing with the tail list q
- $+\!\!+$ is the *append* (list concatenation) operator: for any list l, we have $\mathsf{nil}+\!\!+l \underset{\text{def.}}{=\!=} l$ and $(a :: q)+\!\!+l \underset{\text{def.}}{=\!=} a :: (q+\!\!+l)$.
- $\mathsf{filter}_f(l)$ is the *filter* operator, recursively defined as follows: $\mathsf{filter}_f(\epsilon) = \epsilon$ and $\mathsf{filter}_f(a :: q) = \begin{cases} a :: \mathsf{filter}_f(q) & \text{if } f(a) = \mathsf{true} \\ \mathsf{filter}_f(q) & \text{otherwise} \end{cases}$
- $\mathsf{rev}(l)$ is the *reverse* of the list l, i.e. the list of elements of l given in the reverse order of l. That is, $\mathsf{rev}(\epsilon) = \epsilon$ and $\mathsf{rev}(a :: q) = \mathsf{rev}(q)+\!\!+a :: \epsilon$.
- $\mathsf{length}(l)$ is the length of a list l: $\mathsf{length}(\mathsf{nil}) \underset{\text{def.}}{=\!=} 0$ and $\mathsf{length}(a :: l') \underset{\text{def.}}{=\!=} 1 + \mathsf{length}(l')$

We also adopt the following additional notations:

- $+\!\!+'$ is the operator "append without duplicate", defined as $l_1+\!\!+'l_2 = l_1+\!\!+\mathsf{filter}_{x \mapsto x \notin l_1}(l_2)$
- first is a function defined on non-empty lists, such that $\mathsf{first}(a :: l') \underset{\text{def.}}{=\!=} a$ for all a, l'.
- last is a function defined on non-empty lists, computing their last elements: $\mathsf{last}(a :: \mathsf{nil}) \underset{\text{def.}}{=\!=} a$ and $\mathsf{last}(a :: b :: l') \underset{\text{def.}}{=\!=} \mathsf{last}(b :: l')$ for all a, b, l'.
- for any set $\mathcal{S}$, we pose
$$\mathsf{option}\ \mathcal{S} \underset{\text{def.}}{=\!=} \{\varnothing\} \cup \{\{S\} \mid S \in \mathcal{S}\}$$
which is the set of all subsets of $\mathcal{S}$ with cardinality 0 or 1.

Contents

1 Goal **3**

2 Class hierarchy (reminder) **3**

3 Overview of the construction and destruction process **6**
- 3.1 Construction . . . 6
  - 3.1.1 Non-virtual inheritance only . . . 6
  - 3.1.2 Virtual inheritance . . . 7
- 3.2 Destruction . . . 9

4 Syntax of $\kappa++$ **9**

5 Operational semantics **11**
- 5.1 Construction states . . . 11
- 5.2 Values . . . 14
- 5.3 Execution state . . . 14
  - 5.3.1 Kind . . . 16
  - 5.3.2 Continuation stack . . . 16
- 5.4 Initial and final states . . . 17
- 5.5 Semantic rules . . . 20
  - 5.5.1 Statements . . . 20
  - 5.5.2 Construction . . . 26
  - 5.5.3 Destruction . . . 29

6 Run-Time invariant **32**
- 6.1 Contextual invariants . . . 33
  - 6.1.1 Kind invariant . . . 33
  - 6.1.2 Invariant for stack frames . . . 35
  - 6.1.3 Stackframe chaining . . . 37
  - 6.1.4 Stack well-foundedness . . . 39
- 6.2 Stack objects and constructed stack objects . . . 40
- 6.3 General relations between construction states . . . 42
  - 6.3.1 Vertical relations . . . 42
  - 6.3.2 Horizontal invariant . . . 43

7 Properties of construction and destruction **44**
- 7.1 Progress . . . 45
- 7.2 Increase . . . 46
- 7.3 Construction order . . . 46
  - 7.3.1 Two subobjects of the same complete object . . . 47
  - 7.3.2 Subobjects of different complete objects . . . 50
- 7.4 RAII: Resource Acquisition is Initialization . . . 51
- 7.5 Scalar field access . . . 51
- 7.6 The dynamic type of a subobject . . . 51

8 Conclusion and future work **55**

1 Goal

Our work aims at formalizing an operational semantics for C++ object construction and destruction, and investigating some of its properties.
Our work is machine-checked with the Coq proof assistant.
We reuse the semantics of C++ subobjects defined in our POPL 2011 "object layout" paper. However, now the sets of the direct bases of a class are totally ordered. We denote $\mathcal{D}_C = \{\mathsf{Repeated}\} \times \mathcal{DNV}_C \cup \{\mathsf{Shared}\} \times \mathcal{DV}_C$ the set of all direct bases of C. We denote $\prec^{\mathcal{D}}_C$ the total order on the set of the direct bases of C (this order models the *declaration order*). In the same way, $\prec^{\mathcal{F}}_C$ models the declaration order of the fields of a class.

2 Class hierarchy (reminder)

Notation 1. *Let* $\mathcal{A}$ *be the set of* builtin types *(*$\mathit{int}, \mathit{float}, \mathit{short}, \ldots$*).*

We make no further assumption on builtin types or values: our formalization is a Coq functorial module taking them as parameters.
Contrary to our POPL'11 "Object layout" paper, we pay attention to the *declaration order* of the fields and direct bases of a class.

Definition 1. *A* class hierarchy *is a tuple* $(\mathcal{C}, \mathcal{I}, \mathcal{F}, \mathcal{D})$*, where:*

- $\mathcal{C}$ *is the set of* classes*. We assume* $\mathcal{C} \cap \mathcal{A} = \varnothing$*.*
- $\mathcal{I}$ *is the set of* identifiers
- $\mathcal{F} : \mathcal{C} \rightarrow \mathit{list}(\mathcal{I} \times ((\{\mathit{Sc}\} \times \mathcal{C} \cup \mathcal{A}) \cup (\{\mathit{St}\} \times \mathit{StructField})))$ *gives, for each class, the* ***list*** *of its non-static data members, a.k.a. fields, in declaration order:*
  - *a scalar (*Sc*) field is either a value of a builtin type, or a pointer to an object of some class type* C
  - *a structure array field (*St*)* (f, C, n) *(where* $\mathit{StructField} \overset{\text{def.}}{=\!=} \mathcal{F}I \times \mathcal{C} \times \mathbb{N}^*$*)*[1] *is considered to be an array of* n *structures of type* C*. In fact, structure fields are structure array fields where* $n = 1$*.*
- $\mathcal{D} : \mathcal{C} \rightarrow \mathit{list}(\{\mathit{Repeated}, \mathit{Shared}\} \times \mathcal{C})$ *gives, for each class, the* ***list*** *of its direct bases, either virtual (*Shared*) or non-virtual (*$\mathit{Repeated}$*), in declaration order.*

For instance, the following code:

```
struct A            { int i; };
struct B: virtual A { C * c; };
struct C: A, B      { float j; A a[2]; B b; };
```

would give the following hierarchy:

[1] The Standard forbids arrays with zero cells.

$$
\begin{array}{ccccl}
\mathcal{C} & = & \multicolumn{3}{l}{\{A, B, C\}} \\
\hline
\mathcal{I} & = & \multicolumn{3}{l}{\{i, j, a, b, c\}} \\
\hline
 & & A & \mapsto & (i, (\mathsf{Sc}, \mathsf{int})) :: \epsilon \\
\mathcal{F} & : & B & \mapsto & (c, (\mathsf{Sc}, C)) :: \epsilon \\
 & & C & \mapsto & (j, (\mathsf{Sc}, \mathsf{float})) :: (a, (\mathsf{St}, (A, 2))) :: (b, (\mathsf{St}, (B, 1))) :: \epsilon \\
\hline
 & & A & \mapsto & \varnothing \\
\mathcal{D} & : & B & \mapsto & (\mathsf{Shared}, A) :: \epsilon \\
 & & C & \mapsto & (\mathsf{Repeated}, A) :: (\mathsf{Repeated}, B) :: \epsilon
\end{array}
$$

Notation 2. *We denote $\mathcal{T} = \mathcal{A} \cup \mathcal{C}$ the set of* scalar types*, i.e. builtin types or «pointer to C» for any class C.*

Definition 2. *A list l is a* non-virtual path *from C to A if, and only if:*

- *either $C = A$ and $l = A :: \mathit{nil}$. This path is called the* trivial path.
- *or there exists a non-virtual direct base B of C and a non-virtual path l' from B to A, such that $l = C :: l'$.*

A is a non-virtual base *of C if, and only if, A is reachable through a non-trivial* non-virtual path *from C.*

Definition 3. *A class A is a* virtual base *of C if, and only if:*

- *either A is a direct virtual base of C*
- *or there is a direct (virtual or non-virtual) base B of C such that A is a virtual base of B.*

Definition 4. *A* path, inheritance path, *or* base class subobject, *from a class C to a class A is a couple $\sigma = (h, l)$ such that:*

- *either $h = \mathsf{Repeated}$ and l is a non-virtual path from C to A*
- *or $h = \mathsf{Shared}$ and there exists a virtual base B of C such that l is a non-virtual path from B to A.*

The set of all paths shall be denoted Path. We denote:

$$C \prec\!\langle \sigma \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A$$

the relation «σ is an inheritance path from C to A» inductively defined by the following rules:

$$C \prec\!\langle (\mathsf{Repeated}, C :: \epsilon) \rangle\!\overset{\mathcal{I}}{\rightarrowtail} C$$

$$\frac{B \text{ direct non-virtual base of } C \qquad B \prec\!\langle (\mathsf{Repeated}, l) \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A}{C \prec\!\langle (\mathsf{Repeated}, C :: l) \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A}$$

$$\frac{B \text{ virtual base of } C \qquad B \prec\!\langle (h, l) \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A}{C \prec\!\langle (\mathsf{Shared}, l) \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A}$$

Definition 5. *Let $C, C' \in \mathcal{C}$, $n, n' \in \mathbb{N}$ and α be a list of elements of the set $\mathbb{N} \times \mathsf{Path} \times \mathsf{StructField}$. We say that α* is an array path from $C[n]$ to $C'[n']$ *if, and only if, one of these conditions holds:*

- *$C = C'$ and $n' \leq n$ and $\alpha = \mathsf{nil}$*
- *or all the following conditions hold:*
  - *there exists $i \in \mathbb{N}$ such that $i < n$*
  - *there is a class $A \in \mathcal{C}$ and a path $\sigma \in \mathsf{Path}$ from C to A*
  - *there is a structure array field $F = (f, (\mathsf{St}, (D, s))) \in \mathcal{F}(A)$*
  - *there is a list α' such that α' is an array path from $D[s]$ to $C'[n']$*
  - *and $\alpha = (i, \sigma, F) :: l'$*

We denote:

$$C[n] \prec\!\langle \alpha \rangle\!\overset{\mathcal{A}}{\rightarrowtail} C'[n']$$

defined more formally as:

$$\frac{n' \leq n}{C[n] \prec\!\langle \epsilon \rangle\!\overset{\mathcal{A}}{\rightarrowtail} C[n']} \quad \frac{0 \leq i < n \qquad C \prec\!\langle \sigma \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A \qquad F = (f, (\mathsf{St}, (D, s))) \in \mathcal{F}(A) \qquad D[s] \prec\!\langle \alpha' \rangle\!\overset{\mathcal{A}}{\rightarrowtail} C'[n']}{C[n] \prec\!\langle (i, \sigma, F) :: \alpha' \rangle\!\overset{\mathcal{A}}{\rightarrowtail} C'[n']}$$

Definition 6. *A* generalized subobject *(or simply* subobject*), or* relative pointer *p of static type A relatively to an array of type $C[n]$ (or a* relative pointer from $C[n]$ to A*) is a triple (α, i, σ) where:*

- *α is an array path from $C[n]$ to some $C'[n']$*
- *i is an index in the array of type $C'[n']$, such that $0 \leq i < n'$*
- *and σ is an inheritance path from C' to A*

Then we denote:

$$C[n] \prec\!\langle p \rangle\!\rightarrowtail A$$

and we have, more formally:

$$\frac{0 \leq i < n' \qquad C' \prec\!\langle \sigma \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A}{C' \prec\!\langle (i, \sigma) \rangle\!\overset{\mathcal{CI}}{\rightarrowtail} A} \quad \frac{C[n] \prec\!\langle \alpha \rangle\!\overset{\mathcal{A}}{\rightarrowtail} C'[n'] \prec\!\langle (i, \sigma) \rangle\!\overset{\mathcal{CI}}{\rightarrowtail} A}{C[n] \prec\!\langle (\alpha, i, \sigma) \rangle\!\rightarrowtail A}$$

Definition 7. *A generalized subobject $p = (\alpha, i, \sigma)$ of static type A relatively to an array of type $C[n]$ is a* most-derived object *if, and only if, it cannot be further cast to a derived class. That is, iff $\sigma = (\mathsf{Repeated}, A :: \epsilon)$.*

Definition 8. *A run-time pointer π of static type A is a couple (λ, p) where:*

- *λ is the initial structure array, either statically declared, or dynamically created, of some type $C[n]$ (λ is then called a* complete object*)*
- *p is a relative pointer from $C[n]$ to A*

3 Overview of the construction and destruction process

3.1 Construction

Construction of an object starts when the *constructor* is called with its arguments. An object is constructed when the body of the constructor exits: at this moment starts its *lifetime*.
Construction must follow these two basic principles:

- An object requires prior construction of all of its subobjects
- An object must not be constructed more than once

3.1.1 Non-virtual inheritance only

If a class has no virtual base, then the construction of an instance is straightforward:

- Construct the direct non-virtual bases, in declaration order
- Then construct the fields, in declaration order
- Then run the constructor body

To construct a direct non-virtual base, the constructor of the object first has to compute the arguments to pass to the constructor of the direct non-virtual base. This step corresponds to running the *initializer* for the direct non-virtual base. The construction only starts when the arguments are passed to the constructor.
However, even though arguments are passed, the body of the constructor is not immediately run: the construction of non-virtual bases, and of fields, must first occur.
For instance, consider the following class hierarchy:

```
struct B1 { B1 (int i) {} };
struct B2 { B2 (int j) {} };
struct C: B1, B2 { C (int i1, int i2) : B1 (i1 - i2), B2(i2 - i1) {} }
```

Suppose that an instance of C is requested to be constructed using $C(18, 42)$. Then:

1. Arguments 18 and 42 are passed as i_1 and i_2 to the constructor `C(int, int)`
2. The construction of the non-virtual base B_1 is requested using $B_1(18 - 42)$
3. The value of the argument $18 - 42$ is computed (*initializer* phase)
4. The computed value -24 is passed as i to the constructor `B1(int)`
5. B_1 has no bases or fields, so the body of its constructor is run
6. Then, back to C, the construction of the non-virtual base B_2 is requested using $B_2(42 - 18)$
7. The value of the argument $42 - 18$ is computed (*initializer* phase)
8. The computed value 24 is passed as j to the constructor `B2(int)`

9. B_2 has no bases or fields, so the body of its constructor is run

10. Then, back to C, there are no more bases or fields to construct, so the body of the cosnstructor C is run

To construct a structure array field requires the construction of the object of each array cell, in increasing index order.

3.1.2 Virtual inheritance

Consider the following class hierarchy:

```
struct A {};
struct B0 {};
struct B1: virtual A {};
struct B2: virtual A {};
struct C: B0, B1, B2 {}
```

If A were to be constructed as if it were a non-virtual base, following the above protocol, then A would have been constructed twice. This is prevented by the C++ standard.
In fact, any virtual bases are constructed independently before any non-virtual part of the most-derived object. For instance, in the hierarchy above, the A subobject within C is constructed even before B_0, even though B_0 has no virtual base.
However, this is not precise enough. Indeed, consider the following hierarchy:

```
struct A {};
struct B0 {};
struct B1: virtual A {};
struct B2: virtual A {};
struct C: B0, virtual B1, virtual B2 {}
```

Then, A still has to be prevented from being constructed twice. However, A must be constructed before B_1 and B_2 because A is a base of B_1 and B_2.
In fact, when constructing a most-derived object, all its virtual bases are listed in a certain order $\prec_C^{\mathcal{V}}$ guaranteeing that if a virtual base A of C is actually a virtual base of B, then $A \prec_C^{\mathcal{V}} B$ and A is constructed before B.[2]
Then, to construct a most-derived object of type C, the standard mandates the following process:

- Construct the non-virtual parts of the virtual bases of C, following the order $\prec_C^{\mathcal{V}}$
- Then construct the non-virtual part of the most-derived object

Then, constructing the *non-virtual* part of a subobject follows the protocol mentioned before as if there were no virtual bases.
The order of construction of subobjects is summarized on Figure 1.

[2]The Standard prescribes such an order, called *inheritance graph order*. We shall see further down (48) that we have modeled this order and we have proved that it meets this constraint.

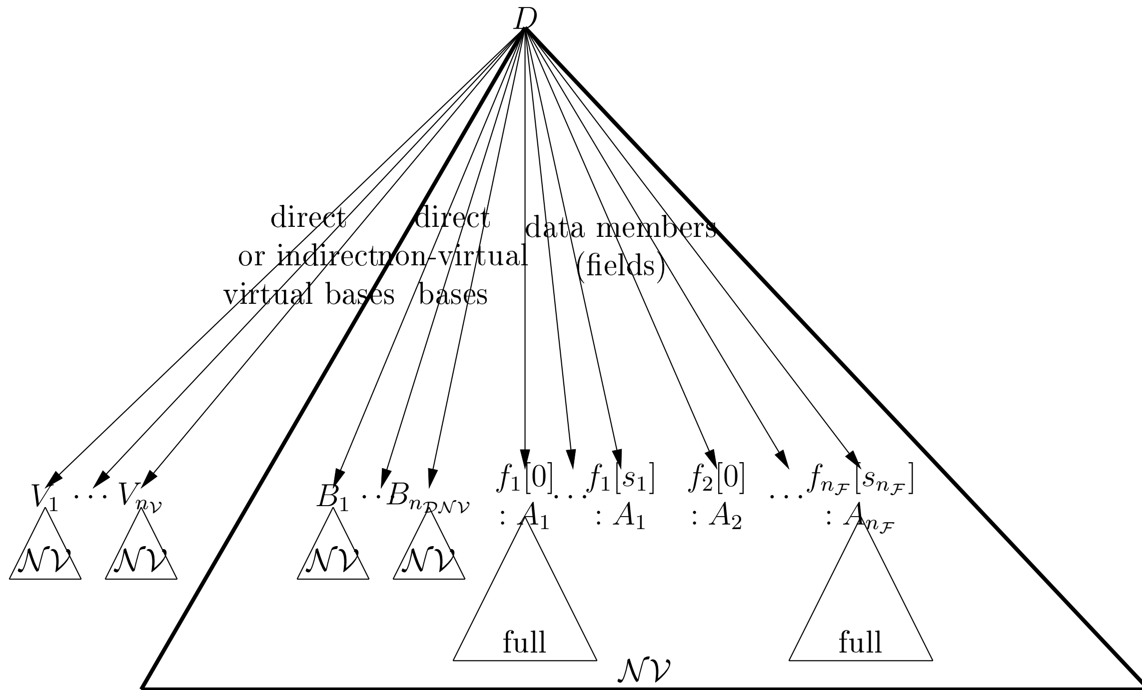


Figure 1: A tree representation of the subobjects of a class, such that a depth-first left-to-right traversal exactly yields the subobject construction order. Class D is assumed to have (direct or indirect) virtual bases $V_1, \ldots, V_{n_\mathcal{V}}$, direct non-virtual bases $B_1, \ldots, B_{n_{\mathcal{DNV}}}$ and structure array fields $f_1 : A_1[s_1], \ldots f_{n_\mathcal{F}} : A_{n_\mathcal{F}}[s_{n_\mathcal{F}}]$. The tree represented here is the "full" tree of all the subobjects of a most-derived object of type D. The subtree inside the thick $\mathcal{NV}$ triangle represents the non-virtual part of D, the only part considered for a D object that is a base-class subobject of another object, thus excluding the virtual bases of D.

3.2 Destruction

Destruction of an object follows the only one principle: if two subobjects constructed in order, then they are destructed in the exact reverse order. In more detail:

- the cells of an array of size n are destructed from cell $n-1$ down to cell 0
- a most-derived object has its non-virtual part destructed first, before its virtual bases following the order $\succ_C^{\mathcal{V}}$ (reverse of $\prec_C^{\mathcal{V}}$)
- to destruct the non-virtual part of an inheritance subobject: the destructor is run first, then the fields are destructed in the reverse declaration order, then the direct non-virtual bases in the reverse declaration order.

4 Syntax of $\kappa++$

The formal model presented in this paper aims at the essence of object construction and destruction semantics. The language we consider features multiple inheritance (both shared and repeated), virtual functions (a.k.a "methods"), nonstatic data members (a.k.a "fields") of any complete non-abstract object type. For the sake of generality and simplicity, a nonstatic data member of type "T" is modeled as a subobject of type "T[1]", e.g. an array of length 1. The expression language is essentially a 3-address language. It supports objects with automatic storage duration, i.e. "stack objects". However, our semantics prohibits any explicit (or manual) object lifetime management. In particular, no form of "new" or "delete" expressions is supported. These restrictions are not as severe as they might sound. Indeed, by and large, the "resource acquisition is initialization" technique involves essentially objects with automatic storage which follow a "stack discipline". Moreover, a large amount of high-level system programs written in C++ do not use "new" or "delete".
Along with a class hierarchy, a program defines virtual functions (or *methods*), constructors and a destructor for each class, each provided with some piece of code.
A virtual function comes with its argument types and its statement body. For the sake of simplicity, virtual function calls return no value.
Function arguments and return values can be either values of builtin types (integers, floats) or pointers to objects: temporary objects for function arguments must be made explicit in our language, and only pointers to structures may be passed. By the way, this allows for fixing the order of construction of temporaries. However, functions returning structures are not allowed.
Each constructor comes with:

- the variable names of its arguments
- the initializers for direct non-virtual bases
- the initializers for fields
- the initializers for all (direct or indirect) virtual bases, used only for the construction of a most-derived object

For any class B, an *initializer* of a subobject of type B is a statement containing a call to a constructor of B with variables given as arguments. Such a statement allows for initializing variables

n	$\in \mathbb{N}$	
$op, \dots$	$: Op$	Atomic operations
$var, \dots$	$: Var$	Variables
$B, C, \dots$	$: Class$	Classes
$fname$	$: Field$	Field names
$mname$	$: Method$	Method names

$Stmt$	$::= var' := op(Var^*)$	Atomic 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
	$\mid var' := var\text{->}_C mname(var^*)$	Virtual function call
	$\mid var' :=$ $\mathtt{static_cast}\langle B\rangle_C(var)$	Static cast
	$\mid Stmt_1; Stmt_2$	Statement sequence
	$\mid \mathsf{if}\ (var)\ Stmt_\top\ \mathsf{else}\ Stmt_\bot$	Conditional
	$\mid \mathsf{loop}\ Stmt$	Loop
	$\mid \{Stmt\}$	Statement block
	$\mid \mathsf{exit}\ n$	Exit from n blocks
	$\mid \mathsf{return}\ var^?$	Return from virtual function
	$\mid \mathsf{skip}$	Do nothing
	$\mid \{C\ var[size] = \{ObjInit^*\}; Stmt\}$	Complete object lifetime
	$\mid C(var^*)$	Constructor call (class initializer only)
	$\mid \mathsf{initScalar}(var)$	Scalar field initialization

Figure 2: Syntax of the core language: statements

for constructor arguments, before handing over to the constructor. A scalar field of a class may also have an initializer, then this initializer exits by giving, through the initScalar statement, the variable used to initialize the field.
Our language only features stack objects declared in scope blocks. It allows no manual memory management: no kind of `new` (neither for dynamic memory allocation, nor for construction at an explicit memory location) or `delete` is provided. Indeed, the lifetime of such objects is more difficult to fit in a RAII (resource acquisition is initialization) model, as it does not necessarily follow a stack discipline. Likewise, our language does not support explicit destructor calls.[3]

In our semantics, an initializer ends by handing over to a constructor. In this last step, it cannot pass a reference to a temporary object. Indeed, such a temporary would have to be destructed after returning from the constructor. Our semantics does not allow initializers to perform any additional steps after calling the constructor.

[3]Thus, our formalization is agnostic on whether destructors should be `virtual` or not

ObjInit	$::= C\{Stmt\}$	Class object initializer
FieldInit	$::= fname\{Stmt\}$	Scalar field initializer
	$\mid fname\{ObjInit^*\}$	Structure field initializers (one for each array cell)
Init	$::= ObjInit \mid FieldInit$	
Constr	$::= C(var^*) : Init^*\{Stmt\}$	Constructor
Destr	$::= \sim C()\{Stmt\}$	Destructor
MethodDef	$::=$ virtual $mname(var^*)\{Stmt\}$	Virtual function (method)
FieldDef	$::=$ scalar $fname;$	Data member
	$\mid$ struct $C[size]\ fname;$	(field)
Base	$::= B \mid$ virtual B	
ClassDef	$::=$ struct $C : Base^*$ $\{FieldDef^*\ MethodDef^*$ $Constr^*\ Destr\}$	Class definition
Program	$::= ClassDef^*;$ main$()\{Stmt\}$	Program

Figure 3: Syntax of the core language: program

In Standard C++, there are only two cases for the initializers of a structure array field f:

- either f has only one cell
- or f has at least two cells, in which case the initializers only call the default constructor (with no argument)

Our semantics widens the Standard C++ by allowing different constructors to be called for each cell.

5 Operational semantics

We formalized a small-step style semantics for C++ object construction and destruction, with a continuation stack to precisely model each step of computation.

5.1 Construction states

Subobjects A constructor body (resp. a destructor) may use **virtual functions** of its class or one of its bases, under the following principle: As long as the body of the constructor (resp. destructor) is still running, the overriding of its virtual functions behaves as if the object being constructed were the most-derived object.
In fact, initializers for fields may also use the virtual functions of the object being constructed, as well as their constructors (indirectly, when a pointer or a reference to the object being constructed

is used within the field constructor). Then, the virtual function overriding mechanism happens "as if" fields were constructed within the body of the constructor. However, virtual functions are prevented from use as long as bases have not been constructed.
To formalize this ability, we introduce the notion of the *construction state* of a subobject. However, this notion has to be interpreted in two different ways, depending on whether the considered object is a most-derived object or an inheritance subobject.
For a most-derived object:

- Unconstructed: Construction has not started yet
- StartedConstructing: The construction of inheritance bases has started, but not the fields
- BasesConstructed: The bases are wholly constructed. Now starting the construction of fields, and the constructor body.
- Constructed: The constructor body has left, and the destruction body has not yet entered
- StartedDestructing: The destructor body has entered, and the fields are being destructed
- DestructingBases: The fields have been wholly destructed. Bases are being destructed
- Destructed: All bases and fields have been destructed

For other inheritance subobjects:

- Unconstructed: Construction of the *non-virtual part* has not started yet (virtual bases may have been already constructed)
- StartedConstructing: The construction of *non-virtual* inheritance bases has started, but not the fields
- BasesConstructed: The bases are wholly constructed. Now starting the construction of fields, and the constructor body.
- Constructed: The constructor body has left, and the destruction body has not yet entered
- StartedDestructing: The destructor body has entered, and the fields are being destructed
- DestructingBases: The fields have been wholly destructed. *Non-virtual bases* are being destructed
- Destructed: All fields and *non-virtual bases* have been destructed

In other words, for an inheritance subobject different from the most-derived object, the construction state is only relative to its *non-virtual part*. This is due to the fact that virtual bases are constructed separately from non-virtual bases: a (virtual or non-virtual) base may have been destructed before its virtual bases.
Then, the *lifetime* of an object can be defined as the time interval when its construction state is exactly Constructed. However, virtual functions may already be used as soon as the construction state is at least BasesConstructed, and strictly before DestructingBases. In that case, for the purpose of function overriding, the object is considered as the most-derived object as long as the construction state is not Constructed.
Consider the following example:

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

Consider an instance of C. Then, during the execution of the constructor body of its base B_2, the corresponding B_2 subobject is BasesConstructed, and the virtual function f can be executed from within the constructor body of B_2.

Definition 9. *We define a successor function[4] S on construction states, such that:*

$$\begin{array}{llllll} & & & & & \mathit{Unconstructed} \\ \stackrel{S}{\mapsto} & \mathit{StartedConstructing} & \stackrel{S}{\mapsto} & \mathit{BasesConstructed} & \stackrel{S}{\mapsto} & \mathit{Constructed} \\ \stackrel{S}{\mapsto} & \mathit{StartedDestructing} & \stackrel{S}{\mapsto} & \mathit{DestructingBases} & \stackrel{S}{\mapsto} & \mathit{Destructed} \end{array}$$

Then, we define an order $<$ on construction states, namely the smallest transitive relation such that $c < \mathsf{S}(c)$.

Fields Similarly, a field can be accessed only if it has been constructed.
We also define the construction state of a scalar field to be one of the following:

- Unconstructed: the field has not yet been constructed, it cannot be accessed yet
- Constructed: the field has been constructed, its value initialized
- Destructed: the field has been destructed, it can no longer be accessed

Similarly, for a structure field:

- Unconstructed: the field has not yet been constructed, it cannot be accessed yet
- StartedConstructing: the cells of the array are under construction
- Constructed: all cells of the array are constructed
- StartedDestructing: the cells of the array are under destruction
- Destructed: all cells of the array are destructed

The differences are that:

- when executing the initializer for a scalar field, it cannot be accessed until given its final value, so it is still considered Unconstructed; however, for a structure array field, as different initializers are used for the different cells of the array, the first cell may be accessed from within the initializer of subsequent cells, so there is an observational difference with Unconstructed which really intends that no subobject of the field has started its construction.
- when destructing a scalar field, the semantics allows no specific code to run (a scalar type has no destructor).

[4]This function is partial, as it is undefined for Destructed

5.2 Values

Notation 3. *We denote $Val_{\mathcal{A}}$ the set of* values of builtin types *(integers, floating-point numbers, booleans, etc.) We denote Loc the set of the locations of complete objects, and Ptr the set of generalized subobjects. Then,*

$$Val = Val_{\mathcal{A}} \cup Ptr$$

is the set of values.

5.3 Execution state

An *execution state* of the small-step semantics is the combination of the following parts:

- the *kind* of the state : code point, or list of objects about to be constructed or destructed
- the *continuation stack* modeling the resumption points on the return from a function, or on the completion of the construction or destruction of a subobject
- the *store*, giving, for each location of a complete object, its class type and array size
- the *scalar field values*
- the *construction states* of subobjects and fields
- the list of *deallocated objects*

For presentation convenience, the store, the scalar field values, the construction states and the list of deallocated objects are grouped into a common *global state*, so that a state is written as a triple $(S, \mathcal{K}, \mathcal{G})$ where S is the kind, $\mathcal{K}$ the continuation stack, and $\mathcal{G}$ the global state grouping the store, the scalar field values, and the construction states.

	Read	Write
Complete object λ is not allocated yet	$\bot = \mathcal{G}.\mathsf{Store}$	N/A
Class type C and array size n of a complete object λ	$(C, n) = \mathcal{G}.\mathsf{Store}(\lambda)$	$\mathcal{G}[\mathsf{Store}(\lambda) \leftarrow (C, n)]$
Value v of the scalar field f of subobject π	$v = \mathcal{G}.\mathsf{FieldValue}(\pi, f)$	$\mathcal{G}[\mathsf{FieldValue}(\pi, f) \leftarrow v]$
Scalar field f of subobject π is not assigned	$\bot = \mathcal{G}.\mathsf{FieldValue}(\pi, f)$	$\mathcal{G}[\mathsf{FieldValue}(\pi, f) \leftarrow \bot]$
Construction state c of a subobject π	$c = \mathcal{G}.\mathsf{ConstrState}(\pi)$	$\mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow c]$
Construction state c of the field f of a subobject π	$c = \mathcal{G}.\mathsf{ConstrState}^{\mathcal{F}}(\pi, f)$	$\mathcal{G}[\mathsf{ConstrState}^{\mathcal{F}}(\pi, f) \leftarrow c]$
List of deallocated objects	$l = \mathcal{G}.\mathsf{dealloc}$	$\mathcal{G}[\mathsf{dealloc} \leftarrow l]$

Notation 4. *Throughout this document, we may also use an alternate notation to read a component of a state $s = (S, \mathcal{K}, \mathcal{G})$: we denote $\mathsf{ConstrState}_s(\pi) = \mathcal{G}.\mathsf{ConstrState}(\pi)$, and similarly for other components of $\mathcal{G}$.*

Once a complete object is given its class type and array size, such data remains in the store forever, to allow reasoning about the construction states of the subobjects even when they are destructed. Thus, the store alone does not say anything about *deallocated* objects. This is the purpose of providing the state with the list of deallocated objects, so that an object λ is allocated if, and only if, its location λ is defined in the store, and is not in the list of deallocated objects.

Definition 10. *(Object lifetime) The* lifetime *of an object (λ, p) is the set of all states $(S, \mathcal{K}, \mathcal{G})$ such that $\mathcal{G}.\mathsf{ConstrState}(\lambda, p) = \mathsf{Constructed}$.*

This notion of lifetime is consistent with the Standard, except for arrays: the (C++03) Standard considers the lifetime of an array to start at allocation and end at deallocation, regardless of the construction and destruction process. Actually, our notion of construction state only covers subobjects with a certain static type, not whole arrays. However, this notion may be fixed in an upcoming C++ Standard to match the array lifetime with the lifetime of its last cell.

Notation 5. *We denote:*

$$\mathcal{G} \vdash (\lambda, p) : B$$

to mean that p is a generalized subobject of the complete object λ, and the static type of p is B. More formally:

$$\frac{\mathcal{G}.\mathsf{Store}(\lambda) = (C, n) \qquad C[n] \prec\!\langle p \rangle\!\succ B}{\mathcal{G} \vdash (\lambda, p) : B}$$

S ::= $\mathsf{Codepoint}(\mathit{Stmt}_1, \mathit{Stmt}^*, \mathit{Env}, \mathit{Block}^*)$

Executing 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 (cf. infra)

| $\mathsf{Constr}(\pi, \mathit{ItemKind}, \kappa, L, \mathit{Env})$

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.

| $\mathsf{ConstrArray}(\lambda, \alpha, n, i, C, \mathit{ObjInit}^*, \mathit{Env})$

About to construct cells i to $n-1$ of type C, of the array α from the complete object λ, 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.

| $\mathsf{DestrArray}(\lambda, \alpha, i, C)$

About to destruct cells i down to 0 of type C, of the array α from the complete object λ

Figure 4: Execution state kind. Cf. auxiliaries Figure 5

5.3.1 Kind

The kind of a state can be of one of the following (Figure 4):

- Executing a statement. This kind also indicates whether the statement is included in a statement block.
- About to construct a list of (virtual or direct non-virtual) bases, or fields
- About to construct an array cell and all its next neighbors
- About to destruct a list of (virtual or direct non-virtual) bases, or fields
- About to destruct an array cell and all its previous neighbors

5.3.2 Continuation stack

A state features a continuation stack to model the pending operations that are to be resumed on the return from a function, or on the completion of the construction or destruction of a subobject. Each element of this stack, or *stack frame* (Figure 6), represents a resumption point. Such a frame can be:

$\mathit{ItemKind}$	$::=$	$\mathsf{Bases}(\mathit{BaseKind})$	Construct (or destruct) bases
	$\mid$	Fields	Construct (or destruct) fields
$\mathit{BaseKind}$	$::=$	$\mathsf{DirectNonVirtual}$	Construct (or destruct) direct non-virtual bases
	$\mid$	$\mathsf{Virtual}$	Construct (or destruct) virtual bases
Block	$::=$	$(\lambda^?, \mathit{Stmt}^*)$	A block: the automatic object to destruct at block exit, if any, and the remaining statements to execute *after* exiting from the block

Figure 5: Execution state kind: auxiliaries

- remaining statements to execute after returning from a function call
- remaining subobjects to construct/destruct
- pending constructor call after returning from the initializer

5.4 Initial and final states

Consider a program of the form

$$\mathit{ClassDef}^*;\ \mathsf{main}()\{\mathit{Stmt}\}$$

Then:

Definition 11. *The initial state is:*

$$(\mathsf{Codepoint}(\mathit{Stmt}, \epsilon, \varnothing, \epsilon), \epsilon, \mathcal{G}_\circ)$$

where:

$$\forall \lambda : \mathcal{G}_\circ.\mathsf{Store}(\lambda) = \bot$$

$$\forall \pi : \mathcal{G}_\circ.\mathsf{ConstrState}(\pi) = \mathsf{Unconstructed}$$

$$\forall \pi, f : \mathcal{G}_\circ.\mathsf{ConstrState}^{\mathcal{F}}(\pi, f) = \mathsf{Unconstructed}$$

$$\forall \pi, f : \mathcal{G}_\circ.\mathsf{FieldValues}(\pi, f) = \bot$$

$$\mathcal{G}_\circ.\mathsf{dealloc} = \epsilon$$

That is, running the main statement with no allocated object at all.

$$\begin{array}{lcl}
\mathcal{K} & ::= & \mathit{Kcode} \\
& \mid & \mathit{Kconstruction} \\
& & \qquad \text{(cf. Figure 7)} \\
& \mid & \mathit{Kdestruction} \\
& & \qquad \text{(cf. Figure 8)}
\end{array}$$

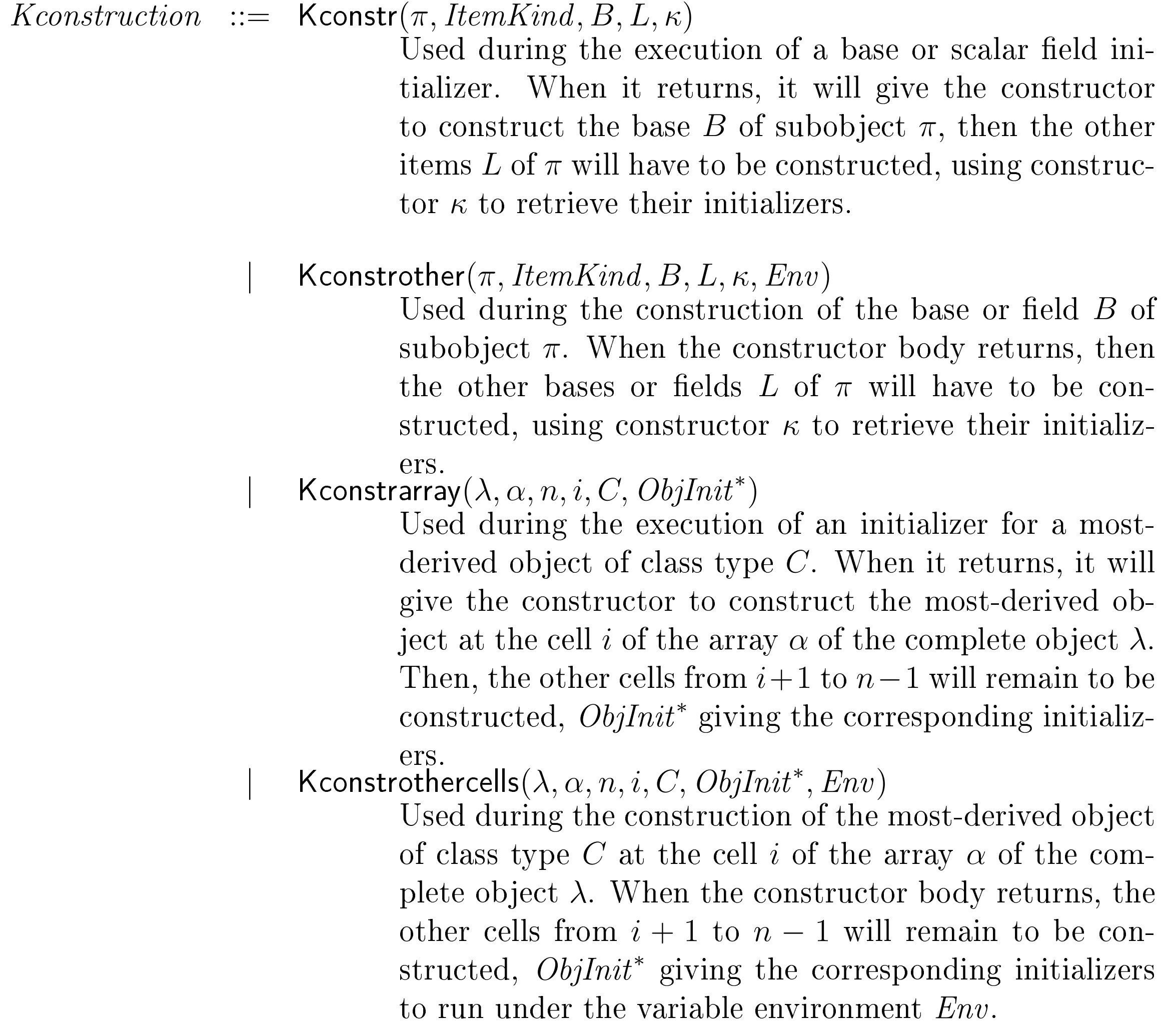
Kcode ::= $\mathsf{Kcontinue}(\lambda, \mathit{Stmt}_1, \mathit{Env}, \mathit{Stmt}_2^*, \mathit{Block}^*)$

After the construction of a complete object λ, enter the block and execute statement Stmt_1, then, after exiting the block, execute statements Stmt_2^* enclosed by other blocks Blocks^*, under variable environment Env. Also used when destructing λ on block exit, with Stmt_1 the corresponding exit statement, and Stmt_2^* the pending statements of the enclosing block, once the block is exited.

| $\mathsf{Kretcall}(\mathit{res}^?, \mathit{Env}, \mathit{Stmt}^*, \mathit{Block}^*)$

On returning from a virtual function call, update the environment Env by storing the result (if any) in variable $\mathit{res}^?$, then continue the caller execution with the further statements Stmt^* enclosed by other blocks Blocks^*.

Figure 6: Continuation stack frames for small-step semantics: code (see also Figure 7, Figure 8)



Kconstruction ::= $\mathsf{Kconstr}(\pi, \mathit{ItemKind}, B, L, \kappa)$

Used during the execution of a base or scalar field initializer. When it returns, it will give the constructor to construct the base B of subobject π, then the other items L of π will have to be constructed, using constructor κ to retrieve their initializers.

| $\mathsf{Kconstrother}(\pi, \mathit{ItemKind}, B, L, \kappa, \mathit{Env})$

Used during the construction of the base or field B of subobject π. When the constructor body returns, then the other bases or fields L of π will have to be constructed, using constructor κ to retrieve their initializers.

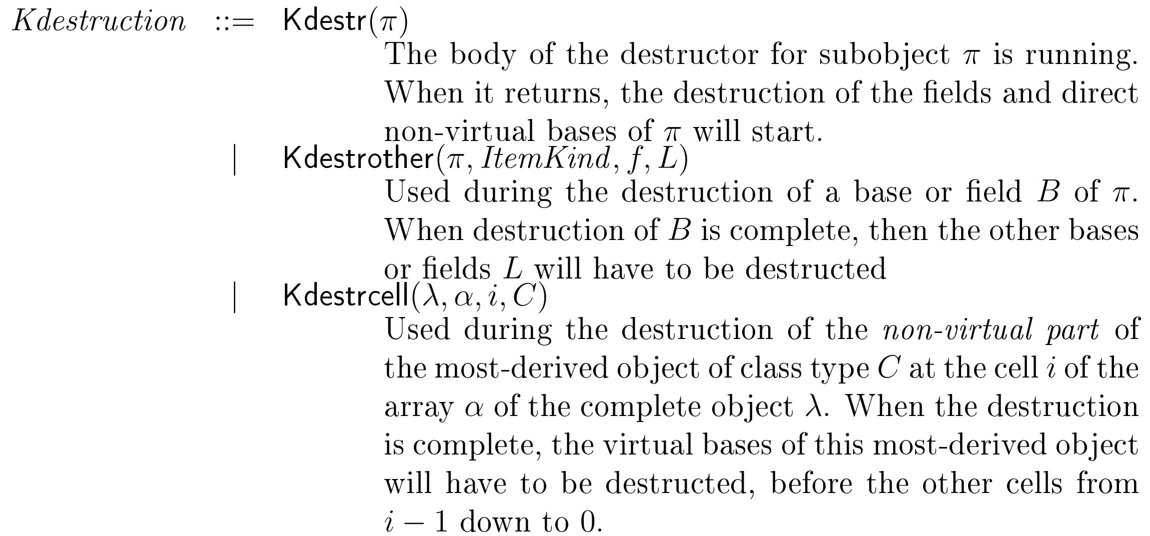
| $\mathsf{Kconstrarray}(\lambda, \alpha, n, i, C, \mathit{ObjInit}^*)$

Used during the execution of an initializer for a most-derived object of class type C. When it returns, it will give the constructor to construct the most-derived object at the cell i of the array α of the complete object λ. Then, the other cells from $i+1$ to $n-1$ will remain to be constructed, $\mathit{ObjInit}^*$ giving the corresponding initializers.

| $\mathsf{Kconstrothercells}(\lambda, \alpha, n, i, C, \mathit{ObjInit}^*, \mathit{Env})$

Used during the construction of the most-derived object of class type C at the cell i of the array α of the complete object λ. When the constructor body returns, the other cells from $i + 1$ to $n - 1$ will remain to be constructed, $\mathit{ObjInit}^*$ giving the corresponding initializers to run under the variable environment Env.

Figure 7: Continuation stack frames for small-step semantics: construction



Kdestruction ::= $\mathsf{Kdestr}(\pi)$

The body of the destructor for subobject π is running. When it returns, the destruction of the fields and direct non-virtual bases of π will start.

| $\mathsf{Kdestrother}(\pi, \mathit{ItemKind}, f, L)$

Used during the destruction of a base or field B of π. When destruction of B is complete, then the other bases or fields L will have to be destructed

| $\mathsf{Kdestrcell}(\lambda, \alpha, i, C)$

Used during the destruction of the *non-virtual part* of the most-derived object of class type C at the cell i of the array α of the complete object λ. When the destruction is complete, the virtual bases of this most-derived object will have to be destructed, before the other cells from $i - 1$ down to 0.

Figure 8: Continuation stack frames for small-step semantics: destruction

Definition 12. *A state* $(S, \mathcal{K}, \mathcal{G})$ *is* final with return value i *if, and only if, all the following conditions hold:*

$$S = \mathsf{Codepoint}(\mathsf{return}\ var, L, Env, \epsilon)$$

$$Env(var) = i \in \mathbb{Z}$$

$$\mathcal{K} = \epsilon$$

That is, the main statement returns with an integer, after having exited from all blocks.

Note that this definition does not a priori prevent from having some undestructed objects in the global state $\mathcal{G}$ of a final state. However, we shall prove that this is not possible (if there is no free store): such a state would not be reachable from an initial state.

5.5 Semantic rules

The small-step semantics of $\kappa++$ is given by the transition relation $\rightarrow$ between two transition states, defined in this section.

5.5.1 Statements

Hypothesis 1. *The set of values of builtin types* $Val_{\mathcal{A}}$ *is assumed to contain:*

- *a subset of* $\mathbb{Z}$ *to model array cell indexes*
- *the Boolean values* true *and* false

In an execution state of kind $\mathsf{Codepoint}(stmt, L, Env, \mathcal{B})$, $stmt$ is the statement to run, and L is a pipeline of pending statements *within the same block* (while $\mathcal{B}$ represents the list of pending enclosing blocks). However, the pipeline is not guaranteed to be executed, in particular if the statement is exit or return.

Structured control The semantics of conditionals depends on the value of its condition variable:

$$\frac{Env(var) = b \ \in \{\mathsf{true}, \mathsf{false}\}}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathsf{if}(var)\ stmt_{\mathsf{true}}\ \mathsf{else}\ stmt_{\mathsf{false}}\ , L, Env, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \rightarrow & (\mathsf{Codepoint}(stmt_b\ , L, Env, \mathcal{B}) \quad\quad\quad\quad ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}} \quad \text{(RULE-if)}$$

The sequence $stmt_1; stmt_2$ runs $stmt_1$, feeding $stmt_2$ into the pipeline.

$$\frac{}{\begin{array}{ll} & (\mathsf{Codepoint}(stmt_1; stmt_2\ , L, Env, \mathcal{B})\ \ ,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \rightarrow & (\mathsf{Codepoint}(stmt_1\ , stmt_2 :: L, Env, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}} \quad \text{(RULE-seq)}$$

The pipeline can be forced by skip:

$$\frac{}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathsf{skip}\ , stmt :: L, Env, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \rightarrow & (\mathsf{Codepoint}(stmt\ , L, Env, \mathcal{B}) \quad\quad ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}} \quad \text{(RULE-skip-cons)}$$

The semantics of a loop is defined by a duplication:

$$\frac{}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathsf{loop}\ stmt\ , L, Env, \mathcal{B}) \quad , \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(stmt\ , \mathsf{loop}\ stmt :: L, Env, \mathcal{B}), \quad \mathcal{K}, \quad \mathcal{G}) \end{array}} \quad \text{(RULE-loop)}$$

The loop is infinite. To break a loop, it is necessary to enclose it in a block. In this section, we first define the semantics of the blocks that do not define stack objects. Entering such a block embeds the pipeline into a new enclosing block added to $\mathcal{B}$ with no stack object.

$$\frac{}{\begin{array}{ll} & (\mathsf{Codepoint}(\{stmt\}\ , L, Env, \mathcal{B}) \quad , \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(stmt\ , \epsilon, Env, (\varnothing, L) :: \mathcal{B}), \quad \mathcal{K}, \quad \mathcal{G}) \end{array}} \quad \text{(RULE-block-no-obj)}$$

exit n exits from n blocks. We first define the semantics of exiting from blocks with no stack objects.

$$\frac{}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathsf{exit}\ 0\ , L, Env, \mathcal{B}), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathsf{skip}\ , L, Env, \mathcal{B}) \quad , \quad \mathcal{K}, \quad \mathcal{G}) \end{array}} \quad \text{(RULE-exit-0)}$$

$$\frac{}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathsf{exit}\ (\mathsf{S}\ n)\ , L, Env, (\varnothing, L') :: \mathcal{B}), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathsf{exit}\ n, L', Env, \mathcal{B}) \quad , \quad \mathcal{K}, \quad \mathcal{G}) \end{array}} \quad \text{(RULE-exit-S)}$$

If there are no more instructions to execute in the block, then the following rule requests automatic exit from the block:

$$\frac{}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathsf{skip}\ , \epsilon, Env, \mathcal{B}) \quad , \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathsf{exit}\ 1\ , \epsilon, Env, \mathcal{B}), \quad \mathcal{K}, \quad \mathcal{G}) \end{array}} \quad \text{(RULE-skip-nil)}$$

This rule implies that if there are no more instructions to execute at the highest level of the function, i.e. $\mathcal{B} = \epsilon$, then the semantics gets stuck. So, in such cases, the user is mandated to explicitly provide a return statement.
We shall see blocks with stack objects in the next section.

Operations on values of builtin types

Notation 6. *For each atomic operation $op \in Op$, we assume there exists a set $\Phi_{op} \subseteq \mathit{list}(Val_{\mathcal{A}}) \times Val_{\mathcal{A}}$ relating operation arguments with the result (which need not exist, nor be unique)*

Then, the atomic operation is modeled as follows:

$$\frac{\forall i, Env(var_i) = v_i \qquad \Phi_{op}(v_1 :: \cdots :: v_n :: \epsilon, res) \qquad Env' = Env[var' \leftarrow res]}{\begin{array}{ll} & (\mathsf{Codepoint}(var' := op(var_1, \ldots, var_n)\ , L, Env, \mathcal{B}), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathsf{skip}\ , L, Env', \mathcal{B}) \quad , \quad \mathcal{K}, \quad \mathcal{G}) \end{array}} \quad \text{(RULE-atom)}$$

Field and array accesses Reading a scalar field is modelled as follows:

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

(RULE-field-scalar-read)

Likewise, writing to a scalar field is modelled as follows:

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

(RULE-field-scalar-write)

Conformingly to the C++ Standard, our formalization forbids writing data to a scalar field that is not Constructed. However, such restrictions are not needed when reading: we can prove that if a field has a value, then it is necessarily Constructed.
Accessing a structure field actually makes a pointer to its first cell, so it is only "pointer adjustment" without actually reading any value. So, as there is no “dereferencing”, no constraint on construction states is needed. However, the complete object must not be deallocated.

$$\frac{Env(var) = \pi = (\lambda, (\alpha, i, \sigma)) \qquad \lambda \notin \mathcal{G}.\mathsf{dealloc} \qquad \mathcal{G} \vdash \pi : C \qquad f = (fid, (\mathsf{St}, B, n)) \in \mathcal{F}(C) \\ Env' = Env[var' \leftarrow (\lambda, (\alpha +\!\!+ (i, \sigma, f) :: \epsilon, 0, (\mathsf{Repeated}, B :: \epsilon)))]}{\begin{array}{ll} & (\mathsf{Codepoint}(var' := var\text{->}_C f\ , L, Env, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \rightarrow & (\mathsf{Codepoint}(\mathsf{skip}\ , L, Env', \mathcal{B}) \qquad\qquad ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}}$$

(RULE-field-struct-point)

Accessing an array cell is only valid on a reference to a most-derived object. Then, again, it is a mere “pointer adjustment” without actually reading any value, so no constraint on construction states is needed. However, the complete object must not be deallocated.

$$\frac{Env(var) = \pi = (\lambda, (\alpha, j, (\mathsf{Repeated}, C :: \epsilon))) \qquad \lambda \notin G.\mathsf{dealloc} \\ \mathcal{G}.\mathsf{Store}(\lambda) = (C', n') \qquad C'[n'] \prec\!\langle \alpha \rangle\!\!\stackrel{\mathcal{A}}{\rightarrowtail} C[n] \qquad 0 \le j < n \qquad Env(var_{\mathsf{i}}) = i \in \mathbb{Z} \\ 0 \le j + i < n \qquad Env' = Env[var' \leftarrow (\lambda, (\alpha, j + i, (\mathsf{Repeated}, C :: \epsilon)))]}{\begin{array}{ll} & (\mathsf{Codepoint}(var' := var[var_{\mathsf{i}}]_C\ , L, Env, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \rightarrow & (\mathsf{Codepoint}(\mathsf{skip}\ , L, Env', \mathcal{B}) \qquad\qquad ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}}$$

(RULE-array-point)

Static cast First, we recall the rules of static cast from B to B' (cf. Wasserrab, Nipkow et al.), denoting $\mathsf{StatCast}(\sigma, \sigma')$ when the cast on inheritance subobject σ succeeds with σ' as the result.

$$\frac{B \prec\!\langle \sigma'' \rangle\!\!\stackrel{\mathcal{I}}{\rightarrowtail} B' \qquad \sigma'' \text{ unique}}{\mathsf{StatCast}(\sigma, B, B', \sigma @ \sigma'')}$$

(RULE-statcast-derived-to-base)

$$\frac{B' \prec\!\langle (\mathsf{Repeated}, B' :: l) \rangle\!\!\stackrel{\mathcal{I}}{\rightarrowtail} B \qquad (\mathsf{Repeated}, B' :: l) \text{ unique}}{\mathsf{StatCast}((h, l' +\!\!+ l), B, B', (h, l'))}$$

(RULE-statcast-base-to-derived-non-virtual)

Then, to reach a base through static cast, the class must have all its bases constructed (construction state between $\mathsf{BasesConstructed}$, and $\mathsf{StartedDestructing}$). From the implementation point of view, it is necessary for the class to know where its bases, in particular its virtual bases, are located, which justifies the requirement on the bases being constructed.

$$\frac{\begin{array}{c} \mathit{Env}(\mathit{var}) = \pi = (\lambda, (\alpha, i, \sigma)) \\ \mathsf{BasesConstructed} \leq \mathcal{G}.\mathsf{ConstrState}(\pi) \leq \mathsf{StartedDestructing} \qquad \mathcal{G}.\mathsf{Heap}(\lambda) = (D, n) \\ D[n] \prec\!(\lambda, i, \sigma)\!\succ\!\!\rightarrow B \qquad \mathsf{StatCast}(\sigma, B, B', \sigma') \qquad \mathit{Env}' = \mathit{Env}[\mathit{var}' \leftarrow \sigma'] \end{array}}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathit{var}' := \mathtt{static_cast}\langle B' \rangle_B(\mathit{var})\ , L, \mathit{Env}, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathsf{skip}\ , L, \mathit{Env}', \mathcal{B})\qquad\qquad\qquad\qquad ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}}$$

(RULE-statcast)

Virtual function call: the dynamic type of a subobject A program is allowed to use virtual functions on a subobject π and all of its bases, in two cases:

- during the lifetime of its most-derived object
- during the execution of the constructor body (or field initializers), or the destructor of π.

But the behaviour of virtual function resolution is not the same in the two cases. Indeed, during construction, the subobject for which the constructor body is running is considered as if it were the most-derived object for the purpose of virtual function resolution. This leads us to define the notion of *generalized dynamic type*, to designate such a subobject, as an extension to the Standard notion of *dynamic type* (which designates the most-derived object for any subobject, during the lifetime of the most-derived object).

In our formalization, a subobject is running its constructor (resp. destructor) body if its construction state is $\mathsf{BasesConstructed}$ (resp. $\mathsf{StartedDestructing}$). So, more formally, we introduce the predicate $\mathcal{G} \vdash \mathsf{gDynType}(\lambda, \alpha, i, \sigma, C_\circ, \sigma_\circ, \sigma')$ to denote that the generalized dynamic type of $(\lambda, (\alpha, i, \sigma))$ is $\sigma_\circ$, which is of static type $C_\circ$, and σ' is an inheritance subobject of $C_\circ$ such that $\sigma = \sigma_\circ @ \sigma'$.

$$\frac{\begin{array}{c} \mathcal{G}.\mathsf{Heap}(\lambda) = (D, n) \\ D[n] \prec\!\alpha\!\succ\!\!\overset{\mathcal{A}}{\rightarrow} C[m] \prec\!(i, \sigma)\!\succ\!\!\overset{\mathcal{CI}}{\rightarrow} B \qquad \mathcal{G}.\mathsf{ConstrState}(\lambda, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) = \mathsf{Constructed} \end{array}}{\mathcal{G} \vdash \mathsf{gDynType}(\lambda, \alpha, i, \sigma, C, (\mathsf{Repeated}, C :: \epsilon), \sigma)}$$

(RULE-dyntype-constructed)

$$\frac{\begin{array}{c} \mathcal{G}.\mathsf{Heap}(\lambda) = (D, n) \qquad D[n] \prec\!\alpha\!\succ\!\!\overset{\mathcal{A}}{\rightarrow} C[m] \prec\!(i, \sigma_\circ)\!\succ\!\!\overset{\mathcal{CI}}{\rightarrow} C_\circ \qquad \mathcal{G}.\mathsf{ConstrState}(\lambda, (\alpha, i, \sigma_\circ)) = c \\ c = \mathsf{BasesConstructed} \lor c = \mathsf{StartedDestructing} \qquad C_\circ \prec\!\sigma'\!\succ\!\!\overset{\mathcal{I}}{\rightarrow} B \qquad \sigma = \sigma_\circ @ \sigma' \end{array}}{\mathcal{G} \vdash \mathsf{gDynType}(\lambda, \alpha, i, \sigma, C_\circ, \sigma_\circ, \sigma')}$$

(RULE-dyntype-pending)

Then, following Wasserrab et al., we denote $\mathsf{VFDispatch}(C_\circ, \sigma', f, B'', \sigma'')$ to say that, if the most-derived object is considered to be of type $C_\circ$, then selecting the dispatch subobject for method f from an inheritance subobject σ' of $C_\circ$ yields the actual subobject σ'' of $C_\circ$, as follows (Figure 9):

- statically choose the static resolving subobject σ_f declaring f (from the static type of σ')
- then choose the *final overrider* for the method. The final overrider is the inheritance subobject σ'' of $C_\circ$ nearest to $C_\circ$ along the path $\sigma' @ \sigma_f$.

$$\frac{B \prec\!\langle \sigma_f \rangle\!\stackrel{\mathcal{I}}{\rightarrowtail} B_f \ni_{\mathsf{method}} f \qquad \forall \sigma_2, B_2 : \begin{array}{c} B \prec\!\langle \sigma_2 \rangle\!\stackrel{\mathcal{I}}{\rightarrowtail} B_2 \ni_{\mathsf{method}} f \Rightarrow \\ \exists \sigma_4 : B_f \prec\!\langle \sigma_4 \rangle\!\stackrel{\mathcal{I}}{\rightarrowtail} \sigma_2 \wedge \sigma_2 = \sigma_f @ \sigma_4 \end{array}}{\mathsf{staticDispatch}(B, f, B_f, \sigma_f)} \quad \text{(RULE-static-dispatch)}$$

$$\frac{\begin{array}{c} C_\circ \prec\!\langle \sigma' \rangle\!\stackrel{\mathcal{I}}{\rightarrowtail} B \qquad \mathsf{staticDispatch}(B, f, B_f, \sigma_f) \qquad C_\circ \prec\!\langle \sigma'' \rangle\!\stackrel{\mathcal{I}}{\rightarrowtail} B'' \ni_{\mathsf{method}} f \qquad B'' \prec\!\langle \sigma''_f \rangle\!\stackrel{\mathcal{I}}{\rightarrowtail} B_f \\ \sigma' @ \sigma_f = \sigma'' @ \sigma''_f \qquad \forall \sigma_2, \sigma_4, B_2 : \begin{array}{c} C_\circ \prec\!\langle \sigma_2 \rangle\!\stackrel{\mathcal{I}}{\rightarrowtail} B_2 \prec\!\langle \sigma_4 \rangle\!\stackrel{\mathcal{I}}{\rightarrowtail} B'' \wedge B_2 \ni_{\mathsf{method}} f \ \Rightarrow \\ B'' = B_2 \end{array} \end{array}}{\mathsf{finalOverrider}(C_\circ, \sigma', f, B'', \sigma'')}$$

(RULE-final-overrider)

$$\frac{\mathsf{finalOverrider}(C_\circ, \sigma', f, B'', \sigma'') \qquad (B'', \sigma'') \text{ unique}}{\mathsf{VFDispatch}(C_\circ, \sigma', f, B'', \sigma'')} \quad \text{(RULE-virtual-dispatch)}$$

Figure 9: Semantics of virtual function call a la Wasserrab, starting from an inheritance subobject σ' of a most-derived object of type $C_\circ$. We denote $B \ni_{\mathsf{method}} f$ the fact that B declares the virtual function f.

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

(RULE-virtual-funcall)

Figure 10: Virtual function call

Finally, we combine our notion of generalized dynamic type with the virtual function dispatch by Wasserrab et al. to obtain our rule for virtual function call:

1. determine the generalized dynamic type $\sigma_\circ$ of $(\lambda, (\alpha, i, \sigma))$; denote $C_\circ$ the type of $\sigma_\circ$, and let σ' such that $\sigma = \sigma_\circ @ \sigma'$
2. dispatch the virtual function assuming that the most-derived object is of type $C_\circ$; denote σ'' the resulting inheritance subobject of $C_\circ$
3. finally the selected subobject is $(\lambda, (\alpha, i, \sigma_\circ @ \sigma''))$, adjust the this pointer to this subobject and call f on it.

We currently do not handle pure virtual functions, or unimplemented virtual functions.
Then, **once all blocks have been exited**, returning from a virtual function call is modelled as follows:

$$\frac{Env(var) = v \qquad Env'' = Env'[res \leftarrow v]}{\begin{array}{lll} & (\mathsf{Codepoint}(\mathsf{return}\ var, L, Env, \epsilon), & \mathsf{Kretcall}(res, Env', L', \mathcal{B}) :: \mathcal{K}, \ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathsf{skip}, L', Env'', \mathcal{B}) & , \qquad \mathcal{K}, \ \ \mathcal{G}) \end{array}} \quad \text{(RULE-return-arg)}$$

$$\frac{}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathsf{return}, L, Env, \epsilon), \quad \mathsf{Kretcall}(\varnothing, Env', L', \mathcal{B}) :: \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathsf{skip}, L', Env', \mathcal{B}), \qquad\qquad\qquad\qquad \mathcal{K}, \quad \mathcal{G}) \end{array}} \quad (\text{RULE-return-no-arg})$$

returning from within a block with no stack objects first dismisses this block:

$$\frac{}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathsf{return}\ var^?, L, Env, (\varnothing, L') :: \mathcal{B}), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathsf{return}\ var^?, L', Env, \mathcal{B}) \qquad\qquad , \quad \mathcal{K}, \quad \mathcal{G}) \end{array}} \quad (\text{RULE-return-block-no-obj})$$

Dynamic cast The behaviour of *dynamic cast* also makes such a distinction on the object that is considered as the most-derived object, "origin" of dynamic cast. So its semantics also makes use of the generalized dynamic type of the subobject.
First, we recall the rules of dynamic cast from B to B' (inspired from Wasserrab et Nipkow) applied to an inheritance subobject σ of C of static type B, considering that the most-derived object is of type C. We denote $\mathsf{DynCast}(C, \sigma, B, B') = \sigma'$ if dynamic cast succeeds, and NULL if it fails. There are four rules:

$$\frac{C \prec\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B \prec\!\langle\sigma''\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B' \qquad \sigma''\ \text{unique}}{\mathsf{DynCast}(C, \sigma, B, B') = \sigma @ \sigma''} \quad (\text{RULE-dyncast-derived-to-base})$$

$$\frac{C \prec\!\langle(h, l')\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B' \prec\!\langle(\mathsf{Repeated}, B' :: l)\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B}{\mathsf{DynCast}(C, (h, l' {+\!\!+} l), B, B') = (h, l')} \quad (\text{RULE-dyncast-base-to-derived-non-virtual})$$

$$\frac{C \prec\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B \qquad C \prec\!\langle\sigma'\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B' \qquad \sigma'\ \text{unique}}{\mathsf{DynCast}(C, \sigma, B, B') = \sigma'} \quad (\text{RULE-dyncast-crosscast})$$

$$\frac{\begin{array}{c} C \prec\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B \qquad \nexists! \sigma'' : B \prec\!\langle\sigma''\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B' \\ \nexists! \sigma' : C \prec\!\langle\sigma'\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B' \qquad \nexists h, l', l : \left\{ \begin{array}{l} \sigma = (h, l' {+\!\!+} l) \\ C \prec\!\langle(h, l')\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B' \prec\!\langle(\mathsf{Repeated}, B' :: l)\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B \end{array} \right. \end{array}}{\mathsf{DynCast}(C, \sigma, B, B') = \mathsf{NULL}}$$

(RULE-dyncast-fail)

Then, the dynamic cast language operation first obtains the generalized dynamic type of the subobject, then performs the cast under this object considered as a most-derived object.

$$\frac{\begin{array}{c} Env(var) = (\lambda, (\alpha, i, \sigma_1)) \qquad \mathcal{G}.\mathsf{Heap}(\lambda) = (D, n) \\ D[n] \prec\!\langle(\lambda, i, \sigma_1)\rangle\!\rightarrowtail B \qquad \mathcal{G} \vdash \mathsf{gDynType}(\lambda, \alpha, i, \sigma_1, C, \sigma_\circ, \sigma) \qquad \mathsf{DynCast}(C, \sigma, B, B') = s \\ s' = \mathsf{match}\ s\ \mathsf{with}\ \sigma' \mapsto (\lambda, (\alpha, i, \sigma_\circ @ \sigma'))\ \mid \mathsf{NULL} \mapsto \mathsf{NULL}\ \mathsf{end} \qquad Env' = Env[var' \leftarrow s'] \end{array}}{\begin{array}{ll} & (\mathsf{Codepoint}(var' := \mathtt{dynamic_cast}\langle B' \rangle_B(var)\ , L, Env, \mathcal{B}), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathsf{skip}\ , L, Env', \mathcal{B}) \qquad\qquad\qquad\qquad , \quad \mathcal{K}, \quad \mathcal{G}) \end{array}}$$

(RULE-dyncast)

5.5.2 Construction

Definition 13. *Let C be a class, and $n \in \mathbb{N}^*$. A* complete *object of type C is an array of structures explicitly declared in a program block, by a C++ language construct of the form:*

$$\{C\ c[n] = \{ObjInit^*\}; Stmt_1\}$$

The notion of *complete* object must not be confused with that of *most-derived* object. Indeed, a *most-derived* object is an object that cannot be cast to a derived class, but it can be a cell of an array field contained in another object. Actually, any most-derived object is a cell of an array field corresponding either to a complete object, or to an object field.
We shall see that, even though *stmt* may return, the destruction of the created block-scoped object is still ensured.
Consider entering a block defining a complete object:

$$\{C\ var[n] = \{inits\}; stmt\}$$

Then, a new object is allocated in the store, and the construction of the array path ϵ starts. The current code point is not yet saved into the list of enclosing blocks, but is still pending in a continuation frame specifying that a block is to be entered.

$$\frac{\lambda \notin \mathsf{dom}(\mathcal{G}.\mathsf{Store}) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{Store}(\lambda) \leftarrow (C, n)] \qquad Env' = Env[c \leftarrow \mathsf{Ptr}(\lambda, \epsilon, (\mathsf{Repeated}, C :: \epsilon))]}{\begin{array}{lll} & (\mathsf{Codepoint}(\{C\ c[n] = \{\iota\}; st\}, St, Env, Bl), & \mathcal{K}, \quad \mathcal{G}) \\ \rightarrow & (\mathsf{ConstrArray}(\lambda, \epsilon, n, 0, C, \iota, Env') \quad , & \mathsf{Kcontinue}(st, Env', St, Bl) :: \mathcal{K}, \quad \mathcal{G}')\end{array}}$$

(RULE-block-obj)

So, when the last cell has been constructed, then the execution resumes, entering a new block (with the detail that the variable environment during array construction supersedes the environment in continuation, as cell initializers may have modified some variables):

$$\frac{}{\begin{array}{lll} & (\mathsf{ConstrArray}(\lambda, \epsilon, n, n, C, \iota, Env') \quad , & \mathsf{Kcontinue}(st, Env, St, Bl) :: \mathcal{K}, \quad \mathcal{G}) \\ \rightarrow & (\mathsf{Codepoint}(st, \epsilon, Env', (\{\lambda\}, St) :: Bl), & \mathcal{K}, \quad \mathcal{G})\end{array}}$$

(RULE-constr-array-nil-kcontinue)

Most-derived object Consider a cell $i < n$ of class type C of array α of size n from a complete object λ. Then, the corresponding initializer is being run, to choose the right constructor for the cell, which is put in a pending state:

$$\frac{i < n \qquad st = \iota(i)}{\begin{array}{lll} & (\mathsf{ConstrArray}(\lambda, \alpha, n, i, C, \iota, Env), & \mathcal{K}, \quad \mathcal{G}) \\ \rightarrow & (\mathsf{Codepoint}(st, \epsilon, Env, \epsilon) \quad , & \mathsf{Kconstrarray}(\lambda, \alpha, n, i, C, \iota) :: \mathcal{K}, \quad \mathcal{G})\end{array}}$$

(RULE-constr-array-cons)

Then, the initializer hands over to a constructor when **there are no pending blocks while running the initializer**. In particular, any object created within the initializer has to be destructed before handing over to the constructor. Thus, no reference to a temporary object can be passed to the constructor. Indeed, such a temporary would have to be destructed after returning from the constructor. Our language does not allow initializers to perform any additional steps after calling the constructor.

So, when the initializer hands over to a constructor to construct an array cell, the arguments are passed to the constructor, forming a new variable environment. Then the construction of a most-derived object starts with the (direct or indirect) virtual bases (assuming the existence of a list $\mathcal{VO}(C)$ of all the virtual bases of a class C, such that, if A and B are virtual bases of C such that A is a virtual base of B, then A appears before B in $\mathcal{VO}(C)$):

$$\frac{\begin{array}{c}\pi = (\lambda, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) \\ L = \mathcal{VO}(C) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{StartedConstructing}] \\ vars = var_0, \dots, var_j \qquad \kappa = C(arg_0, \dots, arg_j) : \dots \{\dots\} \\ \forall i, Env(var_i) = v_i \qquad Env' = \varnothing[arg_0 \leftarrow v_0] \dots [arg_j \leftarrow v_j][\mathsf{this} \leftarrow \mathsf{Ptr}(\pi)]\end{array}}{\begin{array}{ll} & (\mathsf{Codepoint}(C_\kappa(vars), l, Env, \epsilon), \quad \mathsf{Kconstrarray}(\lambda, \alpha, n, i, C, \iota) :: \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), L, \kappa, Env'), \quad \mathsf{Kconstrothercells}(\lambda, \alpha, n, i, C, \iota, Env) :: \mathcal{K}, \quad \mathcal{G}')\end{array}}$$

(RULE-constructor-kconstrarray)

Then, when all virtual bases are done constructing, the construction of the non-virtual part of the object starts, beginning with direct non-virtual bases.

$$\frac{\pi = (\lambda, (\alpha, i, (h, l))) \qquad \mathsf{last}(l) = C \qquad L = \mathcal{DNV}(C)}{\begin{array}{ll} & (\mathsf{Constr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \epsilon, \kappa, Env), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), L, \kappa, Env), \quad \mathcal{K}, \quad \mathcal{G})\end{array}}$$

(RULE-constr-bases-virtual-nil)

For the above rule, the semantics does not *a priori* require that π be a most-derived object when constructing the virtual bases. But in fact, we prove it as a run-time invariant.

Non-virtual part of a subobject For any subobject (that is, not necessarily a most-derived object), after constructing all its direct non-virtual bases, then its fields are being constructed, marking the subobject as BasesConstructed to allow using virtual functions:

$$\frac{\begin{array}{c}\pi = (\lambda, (\alpha, i, (h, l))) \\ \mathsf{last}(l) = C \qquad L = \mathcal{F}(C) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{BasesConstructed}]\end{array}}{\begin{array}{ll} & (\mathsf{Constr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), \epsilon, \kappa, Env), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi, \mathsf{Fields}, L, \kappa, Env), \quad \mathcal{K}, \quad \mathcal{G}')\end{array}}$$

(RULE-constr-bases-direct-non-virtual-nil)

Finally, when all fields have been constructed, the body of the constructor is entered:

$$\frac{\kappa = C(\dots)\{body\}}{\begin{array}{ll} & (\mathsf{Constr}(\pi, \mathsf{Fields}, \epsilon, \kappa, Env), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(body, \epsilon, Env, \epsilon), \quad \mathcal{K}, \quad \mathcal{G})\end{array}}$$

(RULE-constr-fields-nil)

What happens on constructor exit (at a return) depends on the first continuation stack frame, and shall be discussed later.
Now let us see in more detail what happens when constructing a virtual base, a direct (or indirect) non-virtual base, or a field.

Base or scalar field For all cases except structure fields, starting the construction of such a component c first runs the corresponding initializer:

$$\frac{\beta = \mathsf{Fields} \Rightarrow \mathsf{scalar}\ c \qquad \kappa = C(\dots) : \dots, c\{init\}, \dots \{\dots\}}{\begin{array}{ll} & (\mathsf{Constr}(\pi, \beta, c :: L, \kappa, Env), \qquad \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(init, \epsilon, Env, \epsilon)\ ,\ \mathsf{Kconstr}(\pi, \beta, c, L, \kappa) :: \mathcal{K}, \ \mathcal{G}) \end{array}} \quad \text{(RULE-constr-cons)}$$

Then, there are two cases.
First, if c is a scalar field, then the initializer returns by giving, through $\mathsf{init}(var)$, the variable var to be used as the initial value of the field. But again, **all blocks must have been exited before**, no temporaries are allowed to survive to the handover. The given value constructs the field, then other fields are to be constructed.

$$\frac{\mathsf{scalar}\ f \qquad Env(var) = v \qquad \mathcal{G}' = \mathcal{G}[\mathsf{FieldValue}(\pi, f) \leftarrow v][\mathsf{ConstrState}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{Constructed}]}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathsf{init}(var), sl, Env, \epsilon), \ \mathsf{Kconstr}(\pi, \mathsf{Fields}, f, L, \kappa) :: \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi', \mathsf{Fields}, L', Env) \ , \qquad \mathcal{K}, \ \mathcal{G}') \end{array}}$$

(RULE-initscalar)

The second case is if c is a base, that is a (direct or indirect) virtual base, or a direct non-virtual base, then the initializer shall exit through handing over to a constructor, again only with **no pending blocks** (any temporary object has to be destructed before handing over to the constructor). On such an exit, arguments are passed to the constructor (marking the actual start of the construction of the base, thus its construction state changes), then the construction of the **non-virtual part** of the base starts, beginning with its direct non-virtual bases.
The rule below applies for both virtual and direct non-virtual bases, the only difference between the two is the computation of the path of the base.

$$\mathsf{AddBase}((\lambda, (\alpha, i, (h, l))), \beta, B) = \begin{array}{l} \mathsf{match}\ \beta\ \mathsf{with} \\ \mid\ \mathsf{DirectNonVirtual} \ \Rightarrow \ (\lambda, (\alpha, i, (h, l +\!\!+ B :: \epsilon))) \\ \mid\ \mathsf{Virtual} \ \Rightarrow \ (\lambda, (\alpha, i, (\mathsf{Shared}, B :: \epsilon))) \\ \mathsf{end} \end{array}$$

(RULE-addbase)

$$\frac{\begin{array}{c} \pi' = \mathsf{AddBase}(\pi, \beta, B) \qquad \kappa' = B(arg_0, \dots, arg_j) : \dots \{\dots\} \qquad vars = var_0, \dots, var_j \\ \forall i, Env(var_i) = v_i \qquad Env' = \varnothing[arg_0 \leftarrow v_0] \dots [arg_j \leftarrow v_j][\mathsf{this} \leftarrow \mathsf{Ptr}(\pi')] \\ \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi') \leftarrow \mathsf{StartedConstructing}] \end{array}}{\begin{array}{ll} & (\mathsf{Codepoint}(B_{\kappa'}(vars), sl, Env, \epsilon) \ , \ \mathsf{Kconstr}(\pi, \mathsf{Bases}(\beta), B, L, \kappa) :: \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi', \mathsf{Bases}(\mathsf{DirectNonVirtual}), L', Env'), \ \mathsf{Kconstrother}(\pi, \mathsf{Bases}(\beta), B, L, \kappa, Env) :: \mathcal{K}, \ \mathcal{G}') \end{array}}$$

(RULE-constructor-kconstr-base)

Again, for a virtual base, the rule does not explicitly require that the object π be most-derived: we prove it as a run-time invariant.
Then, the construction of the bases goes on, until the constructor body exits through a $\mathsf{return}()$: in that case, the base becomes wholly $\mathsf{Constructed}$, and the construction of other sibling bases goes on.

$$\frac{\pi' = \mathsf{AddBase}(\pi, \beta, B) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi') \leftarrow \mathsf{Constructed}]}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathsf{return}(), \epsilon, Env, \epsilon)\ ,\ \mathsf{Kconstrother}(\pi, \mathsf{Bases}(\beta), B, L, \kappa, Env) :: \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi, \mathsf{Bases}(\beta), L, \kappa, Env), \qquad \mathcal{K}, \ \mathcal{G}') \end{array}}$$

(RULE-return-kconstrother-bases)

A particular case: scalar fields with no initializer In real-world C++, a scalar field may be left uninitialized though declared constructed, if no initializer is specified in the constructor. The following rule allows such a behaviour.

$$\frac{\kappa \text{ has no initializer for } c \qquad c \text{ is scalar} \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrStates}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{Constructed}]}{\begin{array}{ll} & (\mathsf{Constr}(\pi, \mathsf{Fields}, c :: L, \kappa, \mathit{Env}), \quad \mathcal{K}, \quad \mathcal{G}) \\ \rightarrow & (\mathsf{Constr}(\pi, \mathsf{Fields}, L, \kappa, \mathit{Env}) \quad , \quad \mathcal{K}, \quad \mathcal{G}') \end{array}}$$

(RULE-constr-cons-field-scalar-no-init)

Structure fields The construction of a structure field is equivalent to the construction of its array. Contrary to bases, running initializers is part of the construction of the field, so that the construction state of the field changes before the first initializer starts running.

$$\frac{\begin{array}{c} \pi = (\lambda, (\alpha, i, \sigma)) \qquad \mathsf{struct}\ B[n]\ f \\ \kappa = C(\dots) : \dots, f\{\mathit{inits}\}, \dots \{\dots\} \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{StartedConstructing}] \end{array}}{\begin{array}{lll} & (\mathsf{Constr}(\pi, \mathsf{Fields}, f :: L, \kappa, \mathit{Env}) \quad , & \mathcal{K}, \quad \mathcal{G}) \\ \rightarrow & (\mathsf{ConstrArray}(\lambda, \alpha +\!\!+ (i, \sigma, f) :: \epsilon, n, 0, B, \mathit{inits}, \mathit{Env}), & \mathsf{Kconstrother}(\pi, \mathsf{Fields}, f, L, \kappa, \mathit{Env}) :: \mathcal{K}, \quad \mathcal{G}') \end{array}}$$

(RULE-constr-cons-field-struct)

Then, when the last cell of the array is constructed, then the field is considered wholly Constructed, and the construction of other remaining fields goes on, with the variable environment at the end of structure array construction superseding the old one.

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{Constructed}]}{\begin{array}{lll} & (\mathsf{ConstrArray}(\lambda', \alpha', n, n, B, \mathit{inits}, \mathit{Env}'), & \mathsf{Kconstrother}(\pi, \mathsf{Fields}, f, L, \kappa, \mathit{Env}) :: \mathcal{K}, \quad \mathcal{G}) \\ \rightarrow & (\mathsf{Constr}(\pi, \mathsf{Fields}, L, \kappa, \mathit{Env}') \quad , & \mathcal{K}, \quad \mathcal{G}') \end{array}}$$

(RULE-constr-array-nil-kconstrother)

End of the construction of a most-derived object When the body of a most-derived object returns, the first continuation stack frame requests the construction of the further remaining sibling cells of this most-derived object. At that point, the most-derived object becomes Constructed.

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\lambda, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) \leftarrow \mathsf{Constructed}]}{\begin{array}{lll} & (\mathsf{Codepoint}(\mathsf{return}(), l, E', \epsilon) \quad , & \mathsf{Kconstrothercells}(\lambda, \alpha, n, i, C, \iota, \mathit{Env}) :: \mathcal{K}, \quad \mathcal{G}) \\ \rightarrow & (\mathsf{ConstrArray}(\lambda, \alpha, n, i+1, C, \iota, \mathit{Env}), & \mathcal{K}, \quad \mathcal{G}') \end{array}}$$

(RULE-return-kconstrothercells)

5.5.3 Destruction

Complete object We have seen the semantics of exit and return statements when run from inside a block with no stack object.

To exit from a block with a stack object, this object must first be destructed, starting from the destruction of its last cell.

$$\mathit{ExitStmt} ::= \mathsf{exit}\ (\mathsf{S}\ n) \ \mid\ \mathsf{return}\ \mathit{var}^{?} \qquad \text{(RULE-exitstmt)}$$

$$\frac{\mathcal{G}.\mathsf{Store}(\lambda) = (C, n)}{\begin{array}{lll} & (\mathsf{Codepoint}(\mathit{ExitStmt}, L, \mathit{Env}, (\lambda, L') :: \mathcal{B}), & \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{DestrArray}(\lambda, \epsilon, n-1, C), & \mathsf{Kcontinue}(\lambda, \mathit{ExitStmt}, \mathit{Env}, L', \mathcal{B}) :: \mathcal{K}, \quad \mathcal{G}) \end{array}}$$

(RULE-exit-block-obj)

Then, once the cell -1 is requested to be destructed (i.e. once all cells have been destructed), the object is deallocated from the stack (i.e. it disappears from block definitions, and appears in the list of deallocated objects) and the execution resumes after block exit.

$$\begin{array}{rl} \mathsf{ExitSucc}(\mathit{ExitStmt}) = & \mathsf{match}\ \mathit{ExitStmt}\ \mathsf{with} \\ & \mid\ \mathsf{exit}\ (\mathsf{S}\ n) \quad \Rightarrow\ \mathsf{exit}\ n \\ & \mid\ \mathsf{return}\ \mathit{var}^? \quad \Rightarrow\ \mathsf{return}\ \mathit{var}^? \\ & \mathsf{end} \end{array}$$

(RULE-exitsucc)

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{dealloc} \leftarrow \lambda' :: \mathcal{G}.\mathsf{dealloc}]}{\begin{array}{lll} & (\mathsf{DestrArray}(\lambda, \alpha, -1, C), & \mathsf{Kcontinue}(\lambda', \mathit{ExitStmt}, \mathit{Env}, L', \mathcal{B}) :: \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathsf{ExitSucc}(\mathit{ExitStmt}), L', \mathit{Env}, \mathcal{B}), & \mathcal{K}, \quad \mathcal{G}') \end{array}}$$

(RULE-destr-array-nil-kcontinue)

Here, nothing enforces $\lambda = \lambda'$, nor the array path $\alpha = \epsilon$, this is to be shown as an invariant, as we shall see further down.
However, note that $\mathcal{G}.\mathsf{Store}(\lambda)$ is still defined, so as to allow reasoning on construction states even after λ is deallocated. Another purpose of the list of deallocated objects is to prevent from reusing λ for a later allocated object.

Most-derived object When the destruction of a most-derived object (i.e. a structure array cell) is requested, then the destruction of the non-virtual part of this object starts: the destructor body is entered, then the destruction of fields and bases is requested through Kdestr. Kdestrcell reminds, not only that other array cells have to be destructed, but also that a most-derived object is being destructed, so as not to forget virtual bases once the non-virtual part is destructed.

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

(RULE-destr-array-cons)

Non-virtual part of a subobject When a destructor returns, then the fields of the corresponding subobject have to be destructed, in the reverse declaration order.

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

(RULE-return-kdestr)

Destructing a scalar field erases its value and changes its construction state. Then the destruction of other fields is requested.

$$\frac{f = (\mathit{fid}, (\mathsf{Sc}, t)) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{FieldValues}(\pi, f) \leftarrow \bot][\mathsf{ConstrStates}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{Destructed}]}{\begin{array}{ll} & (\mathsf{Destr}(\pi, \mathsf{Fields}, f :: L), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Destr}(\pi, \mathsf{Fields}, L), \quad \mathcal{K}, \quad \mathcal{G}') \end{array}}$$

(RULE-destr-fields-cons-scalar)

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

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

(RULE-destr-fields-cons-struct)

Then, once all cells have been destructed, the field is Destructed, and the destruction of further fields can be proceeded.

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{ConstrStates}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{Destructed}]}{\begin{array}{ll} & (\mathsf{DestrArray}(\lambda', \alpha', -1, C), \quad \mathsf{Kdestrother}(\pi, \mathsf{Fields}, f, L) :: \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Destr}(\pi, \mathsf{Fields}, L), \quad \mathcal{K}, \quad \mathcal{G}') \end{array}}$$

(RULE-destr-array-nil-kdestrother)

Then, once all fields have been destructed, the subobject changes its construction state to DestructingBases(at this point, no virtual function call may be used from this subobject) and the destruction of the direct non-virtual bases can be proceeded, in their reverse declaration order.

$$\frac{\begin{array}{c} \pi = (\lambda, (\alpha, i, \sigma)) \\ \mathsf{last}(l) = C \qquad L = \mathsf{rev}(\mathcal{DNV}(C)) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{DestructingBases}] \end{array}}{\begin{array}{ll} & (\mathsf{Destr}(\pi, \mathsf{Fields}, \epsilon), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), L), \quad \mathcal{K}, \quad \mathcal{G}') \end{array}}$$

(RULE-destr-fields-nil)

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

$$\frac{\begin{array}{c} \sim B()\{\mathit{stmt}\} \\ \pi' = \mathsf{AddBase}(\pi, \beta, B) \qquad \mathit{Env} = \varnothing[\mathsf{this} \leftarrow \pi'] \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi') \leftarrow \mathsf{StartedDestructing}] \end{array}}{\begin{array}{ll} & (\mathsf{Destr}(\pi, \mathsf{Bases}(\beta), B :: L), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathit{stmt}, \epsilon, \mathit{Env}, \epsilon), \quad \mathsf{Kdestr}(\pi') :: \mathsf{Kdestrother}(\pi, \mathsf{Bases}(\beta), B, L) :: \mathcal{K}, \quad \mathcal{G}') \end{array}}$$

(RULE-destr-bases-cons)

Then, once all direct non-virtual bases of π have been destructed, there are two cases, depending on the top of the continuation stack.

Either the continuation stack starts with a $\mathsf{Kdestrother}(\pi', \mathsf{Bases}(\beta))$, then π' is not a most-derived object. So, only the non-virtual part of π had to be destructed, so it may become Destructed, and the destruction of those other bases of π' is requested.

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{StartedDestructing}]}{\begin{array}{ll} & (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), \epsilon), \quad \mathsf{Kdestrother}(\pi', \mathsf{Bases}(\beta), B, L) :: \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Destr}(\pi', \mathsf{Bases}(\beta), L) \quad , \quad \mathcal{K}, \quad \mathcal{G}') \end{array}}$$

(RULE-destr-bases-direct-non-virtual-nil-kdestrother)

End of the destruction of a most-derived object The second case is when the continuation stack starts with a Kdestrcell. Then, π is a most-derived object (this is not constrained by the rules, but must be proved as an invariant), and its virtual bases need to be destructed, in the reverse order of their construction (given by the list $\mathcal{VO}(C)$).

$$\frac{L = \mathsf{rev}(\mathcal{VO}(C))}{\begin{array}{ll} & (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), \epsilon), \quad \mathsf{Kdestrcell}(\lambda, \alpha, i, C) :: \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), L) \quad , \quad \mathcal{K}, \quad \mathcal{G}) \end{array}}$$

(RULE-destr-bases-direct-non-virtual-nil-kdestrcell)

Finally, when all virtual bases have been destructed, then the object (which is actually the most-derived object) is Destructed, and the destruction of further cells may be proceeded.

$$\frac{\pi = (\lambda, (\alpha, i, (h, l))) \qquad \mathsf{last}(l) = C \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{Destructed}]}{\begin{array}{ll} & (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \epsilon), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{DestrArray}(\lambda, \alpha, i-1, C), \quad \mathcal{K}, \quad \mathcal{G}') \end{array}}$$

(RULE-destr-bases-virtual-nil)

Contrary to the construction semantics, one can see that Kdestrcell is not a strict counterpart to Kconstrothercells, as it is not present in the continuation stack when destructing the virtual bases (then, the object π in $\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$ or $\mathsf{Kdestrother}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$ actually refers to the most-derived object from which the next array cell may be deduced. It would have been redundant to keep Kdestrcell under such circumstances; conversely, Kconstrothercells is required during construction because of the variable environment, which changes between each cell due to their initializers.

6 Run-Time invariant

On top of this operational semantics, we built a run-time invariant and we proved that this invariant ever holds during program execution, and also holds for the initial state with no initially allocated objects.
This invariant is built on several layers:

- *Contextual invariants*: states some properties about subobjects involved in the state kind and the stack frames.
  - *Kind-level invariant*: states that the subobject involved in the state kind during construction and destruction are valid (i.e. it is consistent with the hierarchy and the object heap), and sets their construction state with respect to the kind
  - *Stack-level invariant*: states that the subobjects involved in the continuation stack frames are during construction and destruction are valid (i.e. they are consistent with the hierarchy and the object heap), and sets their construction state with respect to the current construction step

  - *Stackframe chaining*: states that the kind requires the presence of a specific frame on top of the stack, and precises similar conditions allowing or not two stack frames to immediately follow each other
  - *Stack well-foundedness*: states that the stack is "sorted" following an order on the subobjects involved in the construction and destruction stack frames

- *Stack objects and constructed stack objects*: shows how to compute the sets of allocated objects and relates their construction states

- *General relations between construction states*:

  - *Vertical invariant*: relates the construction states between an object and its direct (inheritance or field) subobjects (*vertical* relations between construction states)
  - *Horizontal invariant*: relates the construction states between two "sister" subobjects, that is, two subobjects that have the same direct parent object (*horizontal* relations between construction states).

All those invariants trivially hold on the initial state. But, proving their preservation along semantic rules needs around 15000 lines of Coq and around two hours to compile on a Pentium Core Duo 2GHz, consuming around 2Gb of RAM.
Virtually all invariants in this section need each other to hold.
In this whole section, we consider an execution state $(S, \mathcal{K}, \mathcal{G})$.

6.1 Contextual invariants

This section is very technical and allows one to understand how the more "high-level" properties can be proved. It can be skipped at first reading.
The semantic rules are rather lax: they do not appear to constrain the subobject involved in different state kinds or stack frames. However, such conditions are essential to reason about the construction and destruction process.

6.1.1 Kind invariant

To safely apply the semantic rules, some conditions must hold on the subobjects involved in state kinds, and on their construction states.

- Whenever a list of items is requested to be constructed:

$$S = \mathsf{Constr}(\pi, \mathit{ItemKind}, \kappa, L, \mathit{Env})$$

  - π is a valid pointer to a subobject of some static type B (regardless of its construction state):
$$\mathcal{G} \vdash \pi : B$$
  - there is a list L' such that the complete list of *ItemKind* of B can be written as $L' + L$, with any item $A \in L'$ being $\mathsf{Constructed}$, and any item $A \in L$ is $\mathsf{Unconstructed}$.
  - if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$:

    * the construction of π has started, but only concerning its bases:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{StartedConstructing}$$

    * if $\beta = \mathsf{Virtual}$, then π is a most-derived object of static type B
  - otherwise ($\mathit{ItemKind} = \mathsf{Fields}$), the bases of π are already constructed:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{BasesConstructed}$$

- Whenever an array cell is requested to be constructed:

$$S = \mathsf{Constrarray}(\lambda, \alpha, n, i, C, \kappa)$$

  - λ is a valid location of an object in the store:

$$\mathcal{G}.\mathsf{Store}(\lambda) = (C', n')$$

  - α is an array path from (C', n') to (C, n) where n is maximal
  - $0 \leq i \leq n$ (we may have $i = n$, in this case rules (RULE-constr-array-nil-kcontinue, p. 26) and (RULE-constr-array-nil-kconstrother, p. 29) may apply instead of (RULE-constr-array-cons, p. 26))
  - all cells j with $0 \leq j < i$ are Constructed
  - all cells j with $i \leq j < n$ are Unconstructed
- Whenever a list of items is requested to be destructed:

$$S = \mathsf{Destr}(\pi, \mathit{ItemKind}, L)$$

  - π is a valid pointer to a subobject of some static type B (regardless of its construction state):

$$\mathcal{G} \vdash \pi : B$$

  - there is a list L' such that the complete list of $\mathit{ItemKind}$ of B can be written as $\mathsf{rev}(L' + L)$, with any item $A \in L'$ being Destructed, and any item $A \in L$ is Constructed.
  - if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$:
    * the destruction of π has started, already concerning its bases:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{DestructingBases}$$

    * if $\beta = \mathsf{Virtual}$, then π is a most-derived object of static type B
  - otherwise ($\mathit{ItemKind} = \mathsf{Fields}$), the bases of π are still constructed:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{StartedDestructing}$$

- Whenever an array cell is requested to be destructed:

$$S = \mathsf{Destrarray}(\lambda, \alpha, i, C, \kappa)$$

- λ is a valid location of an object in the store:

$$\mathcal{G}.\mathsf{Store}(\lambda) = (C', n')$$

- α is an array path from (C', n') to (C, n) where n is maximal
- $-1 \leq i < n$ (we may have $i = -1$, in this case rules (RULE-destr-array-nil-kcontinue, p. 30) and (RULE-destr-array-nil-kdestrother, p. 31) may apply instead of (RULE-destr-array-cons, p. 30))
- all cells j with $0 \leq j \leq i$ are Constructed
- all cells j with $i < j < n$ are Destructed

The invariant for Codepoint depends on the first item on top of the stack, as we shall see in the next sections.

6.1.2 Invariant for stack frames

To safely apply the semantic rules, some conditions must hold on the subobjects involved in each stack frame, and on their construction states.

- Whenever a list of items is pending to be constructed, during the construction of B:

$$\mathcal{K} \ni \mathsf{Kconstrother}(\pi, \mathit{ItemKind}, \kappa, B, L, \mathit{Env})$$

  - π is a valid pointer to a subobject of some static type B (regardless of its construction state):

  $$\mathcal{G} \vdash \pi : B$$

  - there is a list L' such that the complete list of $\mathit{ItemKind}$ of B can be written as $L' +\!\!+ B :: L$
  - B is StartedConstructing(or maybe BasesConstructed, allowed only if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$)
  - if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$:
    * the construction of π has started, but only concerning its bases:

    $$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{StartedConstructing}$$

    * if $\beta = \mathsf{Virtual}$, then π is a most-derived object of static type B
  - otherwise ($\mathit{ItemKind} = \mathsf{Fields}$), the bases of π are already constructed:

  $$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{BasesConstructed}$$

- Whenever an array cell is requested to be constructed:

$$\mathcal{K} \ni \mathsf{Kconstrothercells}(\lambda, \alpha, n, i, C, \kappa)$$

  - λ is a valid location of an object in the store:

  $$\mathcal{G}.\mathsf{Store}(\lambda) = (C', n')$$

  - α is an array path from (C', n') to (C, n) where n is maximal

- $0 \leq i < n$
- cell i is $\mathsf{StartedConstructing}$, or $\mathsf{BasesConstructed}$

- Whenever a list of items is requested to be destructed:

$$\mathcal{K} \ni \mathsf{Kdestrother}(\pi, \mathit{ItemKind}, B, L)$$

  - π is a valid pointer to a subobject of some static type B (regardless of its construction state):

$$\mathcal{G} \vdash \pi : B$$

  - there is a list L' such that the complete list of $\mathit{ItemKind}$ of B can be written as $\mathsf{rev}(L' {+\!\!+} B :: L)$
  - B is $\mathsf{StartedDestructing}$(or maybe $\mathsf{DestructingBases}$, allowed only if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$)
  - if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$:
    - the destruction of π has started, already concerning its bases:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{DestructingBases}$$

    - if $\beta = \mathsf{Virtual}$, then π is a most-derived object of static type B
  - otherwise ($\mathit{ItemKind} = \mathsf{Fields}$), the bases of π are still constructed:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{StartedDestructing}$$

- Whenever an array cell is requested to be destructed:

$$\mathcal{K} \ni \mathsf{Kdestrcell}(\lambda, \alpha, i, C, \kappa)$$

  - λ is a valid location of an object in the store:

$$\mathcal{G}.\mathsf{Store}(\lambda) = (C', n')$$

  - α is an array path from (C', n') to (C, n) where n is maximal
  - $0 \leq i < n$
  - cell i is $\mathsf{StartedDestructing}$, or $\mathsf{DestructingBases}$

Those invariants are very close to the kind invariants, but they differ in the construction state of the object being constructed or destructed. The construction states of sibling objects are not specified here, but they may be deduced thanks to the horizontal invariants described further down.
Other stack frames only appear in specific contexts: $\mathsf{Kconstr}$ (during the initializer for a base or scalar field), $\mathsf{Kconstrarray}$ (during the initializer for a structure array cell), and Kdestr (during the destructor). For this reason, they are not treated as “stand-alone” stack frames, but separately, in the following section (chaining).

6.1.3 Stackframe chaining

Some stack frames necessarily require to be immediately followed (towards the bottom of the stack) by specific stack frames. The same holds with some kinds, which require specific stack frames on top of the stack. Moreover, depending on those stack frames, their involved subobjects are related in some way.
In this section, we consider that $\mathcal{K} = \mathcal{K}' + K_1 :: K_2 :: \mathcal{K}''$, and we describe, depending on K_1, which frame K_2 may follow. (In parallel, we mention those invariants that may also hold for some kinds requiring that $\mathcal{K} = K_2 :: \mathcal{K}''$).

Construction/destruction kinds and stack frames

- During the construction of the virtual bases of an object π, i.e. any of the following cases:
  - $K_1 = \mathsf{Kconstr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$
  - $K_1 = \mathsf{Kconstrother}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$
  - or $S = \mathsf{constr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

  Then, necessarily, the frame (or kind) is immediately followed by pending cells:

  $$K_2 = \mathsf{Kconstrothercells}(\lambda, \alpha, n, i, C, \dots)$$

  and their subobjects are related:

  $$\pi = (\lambda, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon)))$$

- During the construction of an array cell, i.e. any of the following cases:
  - $K_1 = \mathsf{Kconstrarray}(\lambda, \alpha, i, C, \dots)$
  - $K_1 = \mathsf{Kconstrothercells}(\lambda, \alpha, i, C, \dots)$
  - or $S = \mathsf{constrarray}(\lambda, \alpha, i, C, \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

  Then, necessarily, the frame (or kind) is immediately followed by one of the two cases, depending on α:

  - either $\alpha = \epsilon$: then, the array being constructed is the whole complete object λ itself, so the frame (or kind) must be followed by a code frame $K_2 = \mathsf{Kcontinue}(\lambda, \dots)$ specifying that the array in construction or destruction is actually the allocated stack object.
  - otherwise, $\alpha = \alpha' + (i', \sigma', f') :: \epsilon$ is a structure array field, so the frame (or kind) must be followed by the field construction frame $K_2 = \mathsf{Kconstrother}(\lambda, (\alpha', i', \sigma'), \mathsf{Fields}, f', \dots)$.

- During the construction of the non-virtual bases, or the fields, of an object π, i.e. any of the following cases with $\beta \neq \mathsf{Bases}(\mathsf{Virtual})$:
  - $K_1 = \mathsf{Kconstrother}(\pi, \beta, \dots)$
  - $K_1 = \mathsf{Kconstr}(\pi, \beta, \dots)$
  - or $S = \mathsf{constr}(\pi, \beta, \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

Then, one of the following cases holds, depending on π:

- construction of the bases of some object π':

$$K_2 = \mathsf{Kconstrother}(\pi', \mathsf{Bases}(\beta'), B, \dots)$$

  Then, π is the base B being precisely constructed by K_2:

$$\pi = \mathsf{AddBase}(\pi', \beta', B)$$

- construction of array cells:

$$K_2 = \mathsf{Kconstrothercells}(\lambda, \alpha, i, C)$$

  Then, π is the cell being constructed:

$$\pi = (\lambda, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon)))$$

Similarly, for destruction:

- During the destruction of the virtual bases of an object π, i.e. any of the following cases:
  - $K_1 = \mathsf{Kdestrother}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$
  - or $S = \mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

  or, during the destruction of an array cell, i.e. any of the following cases:

  - $K_1 = \mathsf{Kdestrcell}(\lambda, \alpha, i, C, \dots)$
  - or $S = \mathsf{DestrArray}(\lambda, \alpha, i, C, \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

  Then, necessarily, in the first two cases, $\pi = (\lambda, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon)))$ is a most-derived object; and, in all cases, the frame (or kind) is immediately followed by one of the two cases, depending on α:

  - either $\alpha = \epsilon$: then, the array being destructed is the whole complete object λ itself, so the frame (or kind) must be followed by a code frame $K_2 = \mathsf{Kcontinue}(\lambda, \dots)$ specifying that the array is the stack object.
  - otherwise, $\alpha = \alpha' {+\!\!+} (i', \sigma', f') :: \epsilon$ is a structure array field, so the frame (or kind) must be followed by the field destruction frame $K_2 = \mathsf{Kdestrother}(\lambda, (\alpha', i', \sigma'), \mathsf{Fields}, f', \dots)$.

- During the destruction of the non-virtual part of an object π, i.e. any of the following cases with $\beta \neq \mathsf{Bases}(\mathsf{Virtual})$:
  - $K_1 = \mathsf{Kdestrother}(\pi, \beta, \dots)$
  - $K_1 = \mathsf{Kdestr}(\pi)$ (running the destructor)
  - or $S = \mathsf{destr}(\pi, \beta, \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

  Then, one of the following cases holds, depending on π:

– destruction of the bases of some object π':

$$K_2 = \mathsf{Kdestrother}(\pi', \mathsf{Bases}(\beta'), B, \dots)$$

Then, π is the base B being precisely destructed by K_2:

$$\pi = \mathsf{AddBase}(\pi', \beta', B)$$

– destruction of array cells:

$$K_2 = \mathsf{Kdestrcell}(\lambda, \alpha, i, C)$$

Then, π is the cell being destructed:

$$\pi = (\lambda, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon)))$$

Code points If K_1 is a code frame, that is Kretcall or Kcontinue (or if $S = \mathsf{Codepoint}(\dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$), then K_2 may be one of the following cases:

- another code point, $\mathsf{Kcontinue}(\dots)$ or $\mathsf{Kretcall}(\dots)$
- while executing an initializer for a base or scalar field: $\mathsf{Kconstr}(\pi, \mathit{ItemKind}, B, L, \dots)$. Then, the kind invariant of $\mathsf{constr}(\pi, \mathit{ItemKind}, B :: L)$ holds (the construction states do not change when entering the initializer)
- while executing an initializer for an array cell: $\mathsf{Kconstrarray}(\lambda, \alpha, i, C, \dots)$. Then, the kind invariant of $\mathsf{ConstrArray}(\lambda, \alpha, i, C, \dots)$ holds (the construction states do not change when entering the initializer)
- while executing the constructor for a base B of some object π: $\mathsf{Kconstrother}(\pi, \mathsf{Bases}(\beta), B, L, \dots)$. Then, the base B is BasesConstructed, but not yet Constructed(not before the constructor has exited), so it must be made explicit that all its fields are already Constructed.
- while executing the constructor for a most-derived object (i.e. a structure array cell): $\mathsf{Kconstrothercells}(\lambda, \alpha, n, i, C)$. Then, the cell i is BasesConstructed, but not yet Constructed(not before the constructor has exited), so it must be made explicit that all its fields are already Constructed.
- symmetrically, while executing the destructor for an object: $\mathsf{Kdestr}(\pi)$. Then, π is Started-Destructing, but no longer Constructed, as the destructor entered, so must be made explicit that all fields of π are still Constructed.

6.1.4 Stack well-foundedness

To make reasoning easier, we show an invariant on the stack frames, relating the subobjects of two different stack frames, but regarding the same complete object. Roughly speaking, if (λ, σ) is the object being constructed/destructed by some stack frame, then it is a strict subobject of any object (λ, σ') being constructed/destructed by a stack frame deeper down in the stack.
More precisely:

Definition 14. *The subobject being constructed or destructed by a stack frame K is:*

- *π, if $\mathcal{K}$ is any of* $\mathsf{Kconstr}(\pi, \dots)$, $\mathsf{Kconstrother}(\pi, \dots)$, $\mathsf{Kdestr}(\pi)$ *or* $\mathsf{Kdestrother}(\pi, \dots)$,
- *undefined otherwise*

Similarly, the subobject being constructed or destructed by a state kind S is:

- *π, if S is any of* $\mathsf{Constr}(\pi, \dots)$, $\mathsf{Destr}(\pi, \dots)$
- *undefined otherwise*

Definition 15. *The array being constructed or destructed by a stack frame K is:*

- *(λ, α), if $\mathcal{K}$ is any of* $\mathsf{Kconstrarray}(\lambda, \alpha \dots)$, $\mathsf{Kconstrothercells}(\lambda, \alpha, \dots)$ *or* $\mathsf{Kdestrcell}(\lambda, \alpha, \dots)$
- *undefined otherwise*

Similarly, the array being constructed or destructed by a state kind S is:

- *(λ, α), if S is any of* $\mathsf{ConstrArray}(\lambda, \alpha, \dots)$, $\mathsf{DestrArray}(\lambda, \alpha, \dots)$
- *undefined otherwise*

Then, if $\mathcal{K} = \mathcal{K}' + K_1 :: \mathcal{K}''$, and if $K_2 \in \mathcal{K}''$ (or similarly, if S is a state kind and $K_2 \in \mathcal{K}$), then:

- if $\pi_1 = (\lambda, (\alpha_1, i_1, \sigma_1))$ is the subobject being constructed or destructed by K_1 or S, then:
  - if $\pi_2 = (\lambda, (\alpha_2, i_2, \sigma_2))$ is the subobject being constructed or destructed by K_2, then:
    - either $\alpha_1 = \alpha_2 + \alpha$ for some $\alpha \neq \epsilon$
    - or $\alpha_1 = \alpha_2$, $i_1 = i_2$, and σ_1 is an inheritance subobject of σ_2 distinct from σ_2 itself
  - otherwise, if (λ, α_2) is the array being constructed or destructed by K_2, then $\alpha_1 = \alpha_2 + \alpha$ for some α (maybe ϵ)
- otherwise, if (λ, α_1) is the array being constructed or destructed by K_1 or S, then, if $(\lambda, (\alpha_2, i, \sigma))$ is the subobject, or if (λ, α_2) is the array, being constructed or destructed by K_2, then $\alpha_1 = \alpha_2 + \alpha$ for some $\alpha \neq \epsilon$

6.2 Stack objects and constructed stack objects

In this section, we investigate how to compute the list of stack objects, and stackframe objects, looking at the code points.
When executing a statement block, the block may or may not define a complete object. In this case, this object is called the *stack object* associated to the block. However, when defining such an object, the block receives this object only when it is wholly constructed, and it loses this object once the object starts destruction.
More formally:

Definition 16. *(Stack object of a block) A block $b : Block$ has at most one stack object, denoted $C\Omega(b)$:*

- *If the block is $(\varnothing, Stmt)$, then it has no stack object*
- *If the block is $(\lambda, Stmt)$, then λ is its stack object*

The list of constructed stack objects is computed by gathering all stack objects of all blocks of all code points (code stack frames Kcontinue or Kretcall, and also the kind if it is Codepoint).

Definition 17. *The list of the* constructed stack objects $C\Omega(K)$ *of a stack frame* K *is:*

- *if* $K = \textsf{\textit{Kretcall}}(res^?, Env, Stmt^*, \mathcal{B})$ *or* $K = \textsf{\textit{Kcontinue}}(\lambda^?, Stmt_1, Env, Stmt_2^*, \mathcal{B})$*, then* $C\Omega(K) = \bigcup_{b \in \mathcal{B}} C\Omega(b)$
- *otherwise,* $C\Omega(K) = \epsilon$.

(The notation $\bigcup$ is meant here to keep the order of collected objects following the order of blocks in the block list).

Definition 18. *Similarly, the list of the constructed stack objects* $C\Omega(S)$ *of a state kind* S *is:*

- *if* $S = \textsf{\textit{Codepoint}}(Stmt_1, Stmt^*, Env, \mathcal{B})$*, then* $C\Omega(S) = \bigcup_{b \in \mathcal{B}} C\Omega(b)$
- *otherwise,* $C\Omega(S) = \epsilon$.

Putting all together:

Definition 19. *The list of the constructed stack objects* $C\Omega(S, \mathcal{K})$ *of an execution state* $(S, \mathcal{K}, \mathcal{G})$ *is computed as follows:*

$$C\Omega(S, \mathcal{K}) = C\Omega(S) \cup \bigcup_{K \in \mathcal{K}} C\Omega(K)$$

To collect all stack objects, we must also take into account the objects in construction or destruction, carried by corresponding Kcontinue frames:

Definition 20. *The list of the* stack objects $\Omega(K)$ *of a stack frame* K *are:*

- *if* $K = \textsf{\textit{Kcontinue}}(\varnothing, Stmt_1, Env, Stmt_2^*, \mathcal{B})$ *or* $K = \textsf{\textit{Kretcall}}(res^?, Env, Stmt^*, \mathcal{B})$*, then* $\Omega(K) = \bigcup_{b \in \mathcal{B}} C\Omega(b)$
- *if* $K = \textsf{\textit{Kcontinue}}(\lambda, Stmt_1, Env, Stmt_2^*, \mathcal{B})$*, then* $\Omega(K) = \lambda :: \bigcup_{b \in \mathcal{B}} C\Omega(b)$
- *otherwise,* $\Omega(K) = \epsilon$.

Definition 21. *The list of the stack objects* $\Omega(S, \mathcal{K})$ *of an execution state* $(S, \mathcal{K}, \mathcal{G})$ *is computed as follows:*

$$\Omega(S, \mathcal{K}) = C\Omega(S) \cup \bigcup_{K \in \mathcal{K}} \Omega(K)$$

Lemma 1.

$$C\Omega(S, \mathcal{K}) \subseteq \Omega(S, \mathcal{K})$$

Invariants

- If $\mathcal{G}.\mathsf{Store}(\lambda) = \bot$ is undefined, then any construction state on λ is Unconstructed.
- $\Omega(S, \mathcal{K})$, and $\mathcal{G}.\mathsf{dealloc}$, have no duplicates.
- $\Omega(S, \mathcal{K})$ and $\mathcal{G}.\mathsf{dealloc}$ are disjoint.
- $\mathcal{G}.\mathsf{Store}(\lambda)$ is defined if, and only if, $\lambda \in \Omega(S, \mathcal{K}) \cup \mathcal{G}.\mathsf{dealloc}$
- If $\lambda \in C\Omega(S, \mathcal{K})$ and if $\mathcal{G}.\mathsf{Store}(\lambda) = (C, n)$, then all n cells of λ are Constructed.
- Similarly, if $\mathcal{G}.\mathsf{Store}(\lambda) = (C, n)$ and $\lambda \in \mathcal{G}.\mathsf{dealloc}$, then all n cells of λ are Destructed.

The latter invariant is needed to show the kind invariant of DestrArray for rule (RULE-exit-block-obj, p. 30)

6.3 General relations between construction states

This section is more high-level: it relates the construction states of objects depending on their relative position in the "subobject ordering" tree of Figure 1.

6.3.1 Vertical relations

Definition 22. *Let p, p' be two subobjects of the same complete object. We say that p is a* direct subobject *of p' if either of the following is true:*

- *p is a direct non-virtual base of p'*
- *p' is a most-derived object and p is a virtual base of p'*
- *p is a cell of a structure array field of p' (then, we say that p is a* field subobject *of p').*

In the first two cases, we say that p is a base subobject *of p'.*

By language abuse, we shortly say that p is a *virtual base* (resp. direct non-virtual base) of p' when p designates the subobject corresponding to a virtual base (resp. direct non-virtual base) of the class that is the static type of p'.

Lemma 2 (Middle-level invariant: vertical relations on construction states)**.** *If p is a direct subobject of p', then the following table relates their construction states:*

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

6.3.2 Horizontal invariant

Definition 23. *Let p_1, p_2 be two subobjects of a complete object of type C. We say that p_1* occurs before p_2 *($p_1 \prec_C p_2$) if, and only if, either of the following is true:*

- *there is a most-derived object p' subobject of the complete object C such that p_1, p_2 are two virtual bases of p' in inheritance graph order*
- *there is a subobject p' of the complete object C such that p_1 and p_2 are two direct non-virtual bases of p' in declaration order*
- *p_1 and p_2 are two cells of the same array field, in the order of their indexes within the array*
- *p_1 and p_2 are two cells of two different fields in declaration order*
- *there is a most-derived object p' subobject of the complete object C such that p_1 is a virtual base, and p_2 is a direct non-virtual base of p' or a cell of an array field of p'*
- *there is a subobject p' of the complete object C such that p_1 is a direct non-virtual base of p' and p_2 is a cell of an array field of p'*

Virtual base (if p most-derived)	*Direct non-virtual base*	*Structure array field*
p_1, p_2 *inheritance graph order*		
p_1	p_2	
	p_1, p_2 *declaration order*	
	p_1	p_2
		p_1, p_2 *field declaration order or cell index order of the same field*

This definition is consistent insofar as it only depends on the type of the complete object.

Lemma 3 (High-level invariant: horizontal relations on construction states)**.** *Let p_1, p_2 two subobjects such that $p_1 \prec_C p_2$. Then, the following table relates their construction states:*

If p_1 is...	*Then p_2 is...*
Unconstructed *StartedConstructing* *BasesConstructed*	*Unconstructed*
Constructed	*?*
StartedDestructing *DestructingBases* *Destructed*	*Destructed*

More concisely, for any state s:

$$\mathsf{ConstrState}_s(\lambda, p_1) < \mathsf{Constructed} \Rightarrow \mathsf{ConstrState}_s(\lambda, p_2) = \mathsf{Unconstructed}$$

$$\mathsf{ConstrState}_s(\lambda, p_1) > \mathsf{Constructed} \Rightarrow \mathsf{ConstrState}_s(\lambda, p_2) = \mathsf{Destructed}$$

Corollary 4. *In particular, by contraposition, if two subobjects $p_1 \prec_C p_2$, then the lifetime of p_2 is included in the lifetime of p_1:*

$$\mathsf{ConstrState}_s(\lambda, p_2) = \mathsf{Constructed} \Rightarrow \mathsf{ConstrState}_s(\lambda, p_1) = \mathsf{Constructed}$$

Lemmas 2 and 3 are proved as part of the run-time invariant.

7 Properties of construction and destruction

Thanks to the invariant, we may show a wide range of interesting high-level properties on the construction states of objects during program execution. We first summarize them below:

Theorem 1.

1. *Construction and destruction rules are structurally sound: they may fail only because of wrong user code in initializers or constructor/destructor bodies.*

2. *Construction states are monotonic; each object goes through each construction state following* S *from* Unconstructed *to* Destructed*, and changes its construction state exactly once each. Consequently, each object becomes constructed and destructed exactly once.*

3. *Subobjects of a complete object are destructed in the reverse order of their order. The lifetime of an object is included in the lifetime of all of its subobjects. Moreover, in our semantics, when an object is allocated, other objects allocated before do not change their construction states until the object is deallocated.*

4. *When an object is deallocated, then all its subobjects have been destructed before.*

5. *If a scalar field has a value, then it is constructed.*

6. *The generalized dynamic type of each object and all of its bases changes at well-defined execution points.*

7.1 Progress

To show that our rules "make sense", i.e. that they do not forget any bases to construct, we showed a "sanity-check" *progress* theorem: if a class C and all its children have no user-defined constructors, then the construction and destruction of an instance of C always succeeds.

Definition 24 (Nearly trivial constructor)**.** *We say that a class C has a* nearly trivial constructor *if, and only if, all the following conditions hold:*

- *C has a* default *constructor (with no arguments), having only the following initializers:*
  - *bases and structure array fields are initialized through a call to their default constructor (without arguments), without prior statement*
  - *scalar fields are initialized with constants*[5]*, without prior statement*
- *all virtual bases and direct non-virtual bases of C have nearly trivial constructors*
- *for each structure array field f of C, if f has type B, then B has a nearly trivial constructor*

There is no exactly corresponding notion of the Standard: the latter defines a notion of *trivial constructor* implying that the class C has no virtual bases. In practice, the Standard puts this restriction to allow compilers to produce no code for such classes (e.g. PODs), which is not possible in the presence of virtual bases, as they require additional data (e.g. pointer to virtual tables) to be initialized (cf. our POPL 2011 "object layout" paper, for more accurate information).
However, the "trivial constructor" Standard notion is a particular case of nearly trivial constructors: the conditions are exactly the same, with the further requirements that C have neither virtual bases, nor virtual functions.
In other words, following the terminology of our POPL 2011 "object layout" paper, a class has a trivial constructor if, and only if, it is a non-dynamic class (or, "has no polymorphic behaviour" in the terms of the Standard) with a nearly trivial constructor.

Theorem 2 (Construction progress)**.** *If C is a class having a nearly trivial constructor, then the allocation of a new array of C calling the default constructor for each cell always succeeds, with all cells becoming* Constructed*.*

The same theorem also holds for destruction. But here, we may directly take the Standard notion of *trivial destructor*, as the destructor has no arguments:

[5] or a unique "system call" with constant arguments. The notion of "system call" is defined in CompCert, i.e. an operation taking constant arguments and producing an execution trace, e.g. writing on screen

Definition 25 (Trivial destructor). *A class C has a* trivial destructor *if, and only if, all the following conditions hold:*

- *its destructor immediately* `return`*s*
- *all virtual bases and direct non-virtual bases of C have trivial destructors*
- *for each structure array field f of C, if f has type B, then B has a trivial destructor*

Theorem 3 (Destruction progress). *If C is a class having a trivial destructor, then the deallocation of a constructed array of C always succeeds, with all cells becoming $\mathsf{Destructed}$.*

7.2 Increase

Lemma 5. *If $s \to s'$ is a transition step of the small-step semantics, and if the construction state of (λ, p) goes from c to $c' \neq c$, then $c' = \mathsf{S}(c)$ and any other subobject $p' \neq p$ keeps its construction state unchanged.*

Proof. By case analysis on the small-step semantic rules. □

Corollary 6. *By transitivity, any subobject is never constructed more than once.*

In particular, any *virtual base* subobject is constructed at most once.

Corollary 7. *By contraposition, if $s \to^* s'$ and if $\mathsf{ConstrState}_s(\lambda, p) = \mathsf{ConstrState}_{s'}(\lambda, p)$, then the construction state of (λ, p) remains unchanged between s and s': for any state s'' such that $s \to^* s'' \to^* s'$, we have $\mathsf{ConstrState}_s(\lambda, p) = \mathsf{ConstrState}_{s''}(\lambda, p)$*

In particular, in a given execution sequence, the lifetime of any object is such a state interval.

Corollary 8 (Intermediate values theorem). *If $s \to^* s'$, then, for any subobject (λ, p), and for any construction state c'' such that:*

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

there exist "changing states" s''_1, s''_2 such that:

$$s \to^* s''_1 \to s''_2 \to^* s'$$

and $\mathsf{ConstrState}_{s''_1}(\lambda, p) = c''$ and $\mathsf{ConstrState}_{s''_2}(\lambda, p) = \mathsf{S}(c'')$.
In other words, if an object goes from one construction state to another, then it must go through all construction states in between.

7.3 Construction order

Starting from the run-time invariant described above, we showed some properties about construction order to characterize the notion of RAII (*Resource acquisition is initialization*).
In particular, we aim at showing that destruction of any two subobjects is performed in the reverse order of their construction.

7.3.1 Two subobjects of the same complete object

General theorem Assume that, for any class C and any $n \in \mathbb{N}^*$, there exists a *static* relation $\mathcal{R}_{C[n]}$ (i.e. independent on the execution state) on generalized subobjects of a full object of type $C[n]$, such that for any generalized subobjects p_1, p_2 of $C[n]$, the following conditions hold:

- $p_1 \mathcal{R}_{C[n]} p_2 \vee p_2 \mathcal{R}_{C[n]} p_1$ (i.e. $\mathcal{R}_{C[n]}$ is total)
- for any execution state s, and for any top-level object λ of type $C[n]$, if $p_1 \mathcal{R}_{C[n]} p_2$ and $\mathsf{ConstrState}_s(\lambda, p_2) = \mathsf{Constructed}$, then $\mathsf{ConstrState}_s(\lambda, p_1) = \mathsf{Constructed}$

Then, we may show the following:

Theorem 4. *Let λ be a top-level object of type $C[n]$. If p_1 and p_2 are two generalized subobjects of top-level object λ, such that p_1 was constructed before p_2, then, if p_1 is being destructed, then p_2 was destructed before p_1. In other words, if s_1 is constructed before s_2, then the lifetime of s_2 is included in the lifetime of s_1 .*

$$\begin{array}{ccccccc} s_0 \to s_1 \to^* s_2 \to s_3 & & \to^* & & s_4 \to s_5 \\ \neg c_1 \quad c_1 \quad \neg c_2 \quad c_2 & & & & c_1 \quad \neg c_1 \end{array}$$

$$\Downarrow$$

$$\exists s_3', s_4'$$

$$\begin{array}{c} s_3 \to^* s_3' \to s_4' \to^* s_4 \\ c_2 \quad \neg c_2 \end{array}$$

(c_i means "p_i is $\mathsf{Constructed}$ *at this state")*

Proof. First we show that $p_1 \ \mathcal{R}_{C[n]} \ p_2$. As $\mathcal{R}_{C[n]}$ is total, we may reason by case analysis. Assume $p_2 \ \mathcal{R}_{C[n]} \ p_1$. Then, at state s_1, p_1 is $\mathsf{Constructed}$, so p_2 is also $\mathsf{Constructed}$ at s_1. But construction states are increasing, so p_2 is at least $\mathsf{Constructed}$ at s_2. As it is $\mathsf{Constructed}$ at s_3, it is necessarily $\mathsf{Constructed}$ at s_2, which is absurd. So, necessarily, as $\mathcal{R}_{C[n]}$ is total, we have $p_1 \ \mathcal{R}_{C[n]} \ p_2$.
We immediately see that $p_1 \neq p_2$. Indeed, if $p_1 = p_2$, then, as $\mathsf{Constructed}$ in s_1 and also in s_3, p_2 would also be $\mathsf{Constructed}$ in s_2, which is absurd.
Now we show that p_2 is no longer $\mathsf{Constructed}$ at s_4. As $\leq$ is total on construction states, we may reason by case analysis. Assume p_2 is at most $\mathsf{Constructed}$ at s_4. Then, p_2 is $\mathsf{Constructed}$ at s_4 (increase from s_3). But $p_1 \neq p_2$ and $s_4 \to s_5$ changes the construction state of p_1. Then p_2 is $\mathsf{Constructed}$ also at s_5. But $p_1 \ \mathcal{R}_{C[n]} \ p_2$, so p_1 is also $\mathsf{Constructed}$ at s_5, which is absurd.
To sum up, p_2 is no longer $\mathsf{Constructed}$ at s_4, but it is $\mathsf{Constructed}$ at s_3. So, by the intermediate values theorem, there exists $s_3 \to^* s_3' \to s_4' \to^* s_4$ such that step $s_3' \to s_4'$ makes p_2 from $\mathsf{Constructed}$ to $\mathsf{StartedDestructing}$, which concludes. □

Application It only remains to find such a relation $\mathcal{R}_{C[n]}$. Here we show that the *subobject lifetime relation*, the depth-first left-to-right traversal of the subobject tree of Figure 1, is suitable. Let p_1, p_2 two generalized subobjects of $C[n]$. We say that p_1 is *included* in p_2 (denoted $p_1 \subseteq_{C[n]} p_2$) if, and only if, either $p_1 = p_2$, or there exists a generalized subobject p of $C[n]$ such that p_1 is a direct subobject of p and p is included in p_2.
Roughly speaking, this inclusion relation $\subseteq_{C[n]}$ is the "reflexive and transitive closure" of the "direct subobject" relation. It expresses the notion of path in the subobject tree of Figure 1.
However, this notion is distinct from the notion of inheritance and array paths: if p is not a most-derived object, then it does not include the subobjects corresponding to its virtual bases (only

paths within the *non-virtual part* of the subobject tree are to be considered). Nevertheless, if p is a most-derived object, it does include its virtual bases, so all of its subobjects.
By transitivity, we may show the:

Lemma 9. *If $p_1 \subseteq_{C[n]} p_2$, then the following table relates their construction states:*

If p_2 is...	*Then p_1 is...*
Unconstructed	*Unconstructed*
Constructed	*Constructed*
Destructed	*Destructed*

Finally, we define the *subobject lifetime relation* $\mathcal{R}_{C[n]}$ as follows. Let p_1, p_2 be two subobjects of $C[n]$. We say that p_1 *lays before* p_2, denoted $p_1 \mathcal{R}_{C[n]} p_2$, if, and only if, either condition holds:

- $p_1 \subseteq_{C[n]} p_2$
- there exist two sibling subobjects $p_1' \prec^{\mathcal{D}}_{C[n]} p_2'$ such that $p_1 \subseteq_{C[n]} p_1'$ and $p_2 \subseteq_{C[n]} p_2'$

In fact, this definition says that p_1 lays before p_2 if, and only if, p_1 appears before p_2 in a depth-first left-to-right traversal of the subobject tree. However, we do not need to prove that it is an order.

Lemma 10 ($\mathcal{R}_{C[n]}$ is total)**.**

$$\forall B_1, B_2, p_1, p_2 : \left.\begin{array}{l} C[n] \prec\!\langle p_1 \rangle\!\succ B_1 \\ C[n] \prec\!\langle p_2 \rangle\!\succ B_2 \end{array}\right\} \Rightarrow \left(\begin{array}{l} p_1 \mathcal{R}_{C[n]} p_2 \\ \vee\ p_2 \mathcal{R}_{C[n]} p_1 \end{array}\right.$$

Proof. Long and tedious case analysis. □

Lemma 11. *The subobject lifetime relation is compatible with subobject lifetimes: if $p_1 \mathcal{R}_{C[n]} p_2$, then, for any execution state, and for any top-level object λ of type $C[n]$, whenever (λ, p_2) is Constructed, then (λ, p_1) is Constructed.*

Proof.

- If $p_1 \subseteq_{C[n]} p_2$, then the previous lemma directly applies.
- Otherwise, let $p_1' \prec^{\mathcal{D}}_{C[n]} p_2'$ be two subobjects such that $p_1 \subseteq_{C[n]} p_1'$ and $p_2 \subseteq_{C[n]} p_2'$. We first show that p_1' is Constructed. Let c_1 the construction state of p_1'. As $\leq$ is total on construction states, we have two cases:
  - if $c_1 <$ Constructed, then p_2' is Unconstructed, so p_2 is also Unconstructed, which is absurd.
  - if $c_1 >$ Constructed, then p_2' is Destructed, so p_2 is also Destructed, which is absurd.

  So p_1' is Constructed. Thus, p_1 is also Constructed, which concludes.

□

Subobject ordering and inheritance When constructing the virtual bases of a most-derived object, the Standard prescribes an order called *inheritance graph order*, modelled as follows:

Lemma 12. *If the class hierarchy is well-founded, then the following recursive function $\mathcal{VO}$:*

$$\begin{array}{rcl} \mathit{list}(\{\mathit{Repeated}, \mathit{Shared}\} \times \mathcal{C}) & \stackrel{\mathcal{VO}}{\to} & \mathit{list}(\mathcal{C}) \\ \mathcal{VO}(\epsilon) & = & \epsilon \\ \mathcal{VO}((\mathit{Repeated}, B) :: q) & = & \mathcal{VO}(\mathcal{D}(B)) +\!\!+' \mathcal{VO}(q) \\ \mathcal{VO}((\mathit{Shared}, B) :: q) & = & \mathcal{VO}(\mathcal{D}(B)) +\!\!+' (B :: \mathcal{VO}(q)) \end{array}$$

is well-defined.

$\mathcal{VO}$ actually performs a depth-first search of all virtual bases *induced* by l, including the classes that are elements of l declared as "virtual bases", but quoting each virtual base only once. It is called the *virtual base ordering* function.

Lemma 13. *$\mathcal{VO}(\mathcal{D}(C))$ contains all the virtual bases of C exactly once each, and only them.*

Definition 26. *Two virtual bases A and B of C are in* inheritance graph order*, if, and only if, A occurs before B in $\mathcal{VO}(\mathcal{D}(C))$.*

Lemma 14. *If B is a virtual base of C, then for any virtual base A of B, $A \prec_C^{\mathcal{V}} B$.*

Proof. It suffices to show that A occurs before B in $\mathcal{VO}(\mathcal{D}(C))$. □

Consequently:

Theorem 5. *Let p be a subobject of a complete object $C[n]$. Then, if p' is an inheritance subobject of p, then the lifetime of p is included in the lifetime of p'.*

Proof. There are two cases:

- If p' is a non-virtual base-class subobject of p, then $p' \subseteq_{C[n]} p$, so a previous lemma applies.
- Otherwise, if p' is a virtual base-class subobject of p, then there are two cases:
  - If p is a most-derived object, then $p' \subseteq_{C[n]} p$.
  - Otherwise, we can show that $p' \mathcal{R}_{C[n]} p$. Let A be the static type of p. Then, p' is a non-virtual base-class subobject of some virtual base V of A. There are two cases:
    * If p is a non-virtual base-class subobject of its most-derived object, then it is a non-virtual base-class subobject of some base B; so let p_V and p_B represent the direct subobjects of the (common) most-derived object of p and p' for B and V, so that $p' \subseteq_{C[n]} p_V$ and $p \subseteq_{C[n]} p_B$. As B is a non-virtual base and V is a virtual base, then we have $p_V \prec_{C[n]} p_B$, which concludes.
    * Otherwise, p is a virtual base-class subobject of its most-derived object, then it is a non-virtual base-class subobject of some virtual base B of the most-derived object. By transitivity, V is a virtual base of B, so $p_V \prec_{C[n]} p_B$ by the above lemma, and we can conclude similarly as the previous case.

□

Lemma 15. *If $p_1 \subseteq_{C[n]} p_2 \mathcal{R}_{C[n]} p_3$, then $p_1 \mathcal{R}_{C[n]} p_3$.*

Proof. By definition of $\mathcal{R}_{C[n]}$ and by transitivity of $\subseteq_{C[n]}$. □

Theorem 6. *Let p be a subobject of a complete object $C[n]$. Then, if p' is a subobject of p, then the lifetime of p is included in the lifetime of p'.*

Proof. There are two cases:

- If p' is an inheritance subobject of p, then the previous lemma applies.
- Otherwise, $p' \subseteq p_f \subseteq p_B$ where p_f is an array cell of some field f of some inheritance subobject of p_B of p, so we may conclude by the two lemmas above.

□

7.3.2 Subobjects of different complete objects

In general, in a real-world C++ program (except for embedded systems, where dynamic memory allocation is not necessarily permitted), there is no information about whether two complete objects created by `new` have their lifetimes included, disjoint or overlapping.
However, in our model where all objects are in stack, we may prove some kind of *stack discipline* for object lifetimes.

Lemma 16. *Consider an object λ. If it is defined in the store:*

$$\mathcal{G}.\mathsf{Store}(\lambda) = (C, n)$$

but outside the list $\Omega(S, \mathcal{K})$ of stack objects, then all n cells of λ are Destructed*.*

Proof. This can be proved as an additional run-time invariant. It needs, however, the run-time invariant about the precise construction states of objects (kind invariants): for the particular step (RULE-destr-array-nil-kcontinue, p. 30) when an object is about to be deallocated, this object must be Destructed. ☐

Hypothesis 2. *We assume a total order over object locations, such that the operation "retrieve a new fresh location in the object store" be* strictly increasing.

In practice, object locations may range over $\mathbb{Z}$, for instance. This is the case in our Coq development.

Lemma 17. *If $s \to s'$ is a step changing the construction state of a subobject (λ, p), then there can be no object $\lambda' > \lambda$ in the set of allocated objects.*

Proof. It suffices to show that the set of allocated objects forms an *ordered stack* w.r.t. $<$. This can be proved as an invariant along with the run-time invariant.
Then, operations over construction states only modify the top-most object of this stack, which is maximal w.r.t. $<$. ☐

Theorem 7. *If $s \to^* s'$ and if λ is a complete object belonging to the set of allocated objects for states s and s', then, for any subobject (λ', p') such that $\lambda' < \lambda$, the construction state of (λ', p') does not change between s and s'.*

Proof. Follows from the above lemma, by transitivity. ☐

Lemma 18. *If, between s and s', an object in the allocation set of s is no longer in the allocation set of s', then it is deallocated between s and s'.*

Proof. Trivial induction on the length of the execution path $s \to^* s'$. ☐

Theorem 8. *Let λ be an allocated object, and $s_0 \to s$ be an allocation step of some object $\lambda' \neq \lambda$. Then, if $s \to^* s'$ and if (λ, p) changes its construction state between s and s', then λ' is deallocated between s and s'.*

Proof. The small-step semantic rule for object allocation $s_0 \to s$ only allocates λ', with λ already allocated, so $\lambda' > \lambda$.
Let c be the construction state of (λ, p) at s. Then, by the intermediate values theorem, there exists $s \to^* s_1 \to s_2 \to^* s'$ such that $s_1 \to s_2$ makes (λ, p) from c to $\mathsf{S}(c)$. At this step, λ is necessarily the top-most object on the allocation stack, so in particular, at s_1, λ' is no longer allocated. So, by the previous lemma, there exists a deallocation step for λ' between s and s_1. ☐

7.4 RAII: Resource Acquisition is Initialization

Theorem 9. *If an object is deallocated, then it has been constructed and destructed before, in this order.*

Proof. Let $s \to s'$ be the deallocation step of an object λ. Then, by the low-level invariant, we know that the construction state of λ is Destructed at s, so is it for any subobject p of λ. As it is Unconstructed at the initial state of the program, then, by the intermediate values theorem, (λ, p) passes through a step Constructed $\to$ StartedDestructing, which corresponds to entering the destructor. Again, before this step, (λ, p) passes through a step BasesConstructed $\to$ Constructed, which corresponds to leaving the constructor body. □

Theorem 10. *At the end of the program, all objects were destructed.*

Proof. It suffices to show that at the final step, there are no allocated objects. This can be shown thanks to the structure of the final step. The result then follows from the above lemma. □

Again, this theorem is not true in the presence of a free store (nothing guarantees that `delete` has been called for each dynamically allocated object).

7.5 Scalar field access

Two properties about scalar field accesses may be easily shown as an invariant:

Lemma 19. *If a scalar field has a value, then it is Constructed.*

Proof. There are only three rules modifying the value of a field:

- (RULE-field-scalar-write, p. 22) explicitly requires the field being Constructed
- (RULE-initscalar, p. 28), giving the field its initial value, switches the field construction state to Constructed
- (RULE-destr-fields-cons-scalar, p. 31) erases the value of the field, so the hypothesis no longer holds

However, the run-time invariant is needed to discriminate between a scalar and a structure field when its construction state changes. □

Analogously:

Lemma 20. *If rule (RULE-constr-cons-field-scalar-no-init, p. 29) is disabled, then a field has a value if, and only if, it is Constructed.*

7.6 The dynamic type of a subobject

In fact, we aim at showing that, for any most-derived object, there is at most one inheritance subobject that can play the role of generalized dynamic type for a given execution state.
So, tailoring rules (RULE-dyntype-constructed, p. 23) and (RULE-dyntype-pending, p. 23), we can redefine the notion of generalized dynamic type only depending on the most-derived object. By language abuse, we say that σ is the *generalized dynamic type of the structure array cell* (λ, α, i) and we denote getGenDynType$(\lambda, \alpha, i, \sigma)$:

$$\frac{\mathcal{G}.\mathsf{Heap}(\lambda) = (D, n) \quad D[n] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\rightarrowtail} C[m] \quad \mathcal{G}.\mathsf{ConstrState}(\lambda, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) = \mathsf{Constructed}}{\mathcal{G} \vdash \mathsf{getGenDynType}(\lambda, \alpha, i, (\mathsf{Repeated}, C :: \epsilon))}$$

(RULE-getdyntype-constructed)

$$\frac{\mathcal{G}.\mathsf{Heap}(\lambda) = (D, n) \quad D[n] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\rightarrowtail} C[m] \prec\!\langle(i, \sigma')\rangle\!\overset{\mathcal{CI}}{\rightarrowtail} B' \quad \mathcal{G}.\mathsf{ConstrState}(\lambda, (\alpha, i, \sigma')) = c \quad c = \mathsf{BasesConstructed} \vee c = \mathsf{StartedDestructing}}{\mathcal{G} \vdash \mathsf{gDynType}(\lambda, \alpha, i, \sigma')}$$

(RULE-getdyntype-pending)

Immediately, we then have that:

Lemma 21. *Let λ be a complete object of type $D[n]$, and α such that $D[n] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\rightarrowtail} C[m]$ and $0 \leq i < m$. Let σ' be an inheritance subobject of C of static type B. Then, if σ' is the generalized dynamic type of the structure array cell (λ, α, i), then, for any inheritance subobject σ'' of B, σ' is the generalized dynamic type of the subobject $(\lambda, (\alpha, i, \sigma'@\sigma''))$:*

$$\textit{getGenDynType}(\lambda, \alpha, i, \sigma') \Rightarrow \textit{gDynType}(\lambda, \alpha, i, \sigma'@\sigma'', B, \sigma', \sigma'')$$

Lemma 22. *Conversely, for any inheritance subobject σ of C, if σ' is the generalized dynamic type of $(\lambda, (\alpha, i, \sigma))$ such that $\mathsf{gDynType}(\lambda, \alpha, i, \sigma, B, \sigma', \sigma'')$ for some B and σ'', then σ' is the generalized dynamic type of the array cell (λ, α, i) and there is an inheritance subobject σ'' of σ' such that $\sigma = \sigma'@\sigma''$.*

$$\textit{gDynType}(\lambda, \alpha, i, \sigma, B, \sigma', \sigma'') \Rightarrow \textit{getGenDynType}(\lambda, \alpha, i, \sigma') \wedge \sigma = \sigma'@\sigma''$$

In other words, the generalized dynamic type can be obtained using the getGenDynType predicate, whereas gDynType can be used to determine whether a subobject has its generalized dynamic type defined, and how the corresponding inheritance subobject of the generalized dynamic type can be deduced.
Moreover, an inheritance subobject has its generalized dynamic type defined only if it is a base of the generalized dynamic type of the array cell. Indeed, consider the following example:

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

Consider an instance of C. Then, during the execution of the constructor body of its base B_2, the corresponding B_2 subobject is BasesConstructed, so it is the generalized dynamic type of the array cell. But, even though the subobject B_1 is already Constructed, its generalized dynamic type is undefined, as B_1 is not a base of B_2. So, calling f on B_1 has undefined behaviour.
Now, we can reason about the generalized dynamic type of an array cell instead of considering the generalized dynamic type of a subobject.

Lemma 23. *Considering a most-derived object, there can be at most one* inheritance *subobject in construction state BasesConstructed or StartedDestructing.*

Proof. If there are two of them, say p_1 and p_2, then there are two cases:

- say p_2 is a base of p_1. Then, as p_1 is BasesConstructed or StartedDestructing, all its bases are Constructed, in particular p_2, which is absurd.
- otherwise, there is a subobject p and two direct bases p'_1, p'_2, say in this order, such that each p_i is a base of p'_i. Then, p'_2 is Unconstructed or Destructed, so p_2 as well, which is absurd.

□

Corollary 24. *The generalized dynamic type of an array cell, if any, is unique.*

Proof. If the most-derived object is Constructed, then the result is trivial. Otherwise, it follows from the above lemma. □

However, the generalized dynamic type of an array cell does not *continuously* exist: during the lifetime of the subobject, while the most-derived object is not yet Constructed, the generalized dynamic type of the array cell does exist only if there is an inheritance subobject that is BasesConstructed or StartedDestructing.
Indeed, in the above example, after exiting from the body of the constructor for B_1, but before entering the body of the constructor for B_2, there is no constructor body in progress for the instance of C, so there is no generalized dynamic type for the instance C.

Lemma 25. *The following table summarizes the evolution of the dynamic type of an array cell* (λ, α, i) *of type* C *(denoting* $\sigma_\circ = (\mathsf{Repeated}, C :: \epsilon)$ *the corresponding most-derived object)*

When the subobject	*goes from*	*to*	*then the dynamic type of* (λ,α,i) *goes* *from*	*to*
$(\lambda, (\alpha, i, \sigma))$	*Unconstructed*	*StartedConstructing*	*Undefined*	*Undefined*
$(\lambda, (\alpha, i, \sigma))$	*StartedConstructing*	*BasesConstructed*	*Undefined*	σ
$(\lambda, (\alpha, i, \sigma))$ *with* $\sigma \neq \sigma_\circ$	*BasesConstructed*	*Constructed*	σ	*Undefined*
$(\lambda, \alpha, i, \sigma_\circ)$	*BasesConstructed*	*Constructed*	$\sigma_\circ$	$\sigma_\circ$
$(\lambda, \alpha, i, \sigma_\circ)$	*Constructed*	*StartedDestructing*	$\sigma_\circ$	$\sigma_\circ$
$(\lambda, (\alpha, i, \sigma))$ *with* $\sigma \neq \sigma_\circ$	*Constructed*	*StartedDestructing*	*Undefined*	σ
$(\lambda, \alpha, i, \sigma)$	*StartedDestructing*	*DestructingBases*	σ	*Undefined*
$(\lambda, \alpha, i, \sigma)$	*DestructingBases*	*Destructed*	*Undefined*	*Undefined*
$(\lambda', (\alpha', i', \sigma'))$ *with* $(\lambda, \alpha, i) \neq (\lambda', \alpha', i')$	*Any*	*Any*	*Does not change*	

Corollary 26. *The following table summarizes the evolution of the dynamic types of a subobject*

depending on the evolution of construction states.

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

In more detail:

Lemma 27. *When a subobject p becomes BasesConstructed or StartedDestructing:*

- *its dynamic type changes and becomes defined, as well as the dynamic type of all of its bases.*
- *the dynamic type of all other subobjects (which are not bases of p) cannot change to a defined value.*

Proof.

- The first case is obvious, as the dynamic type cannot be p before it becomes BasesConstructed.
- In the second case, consider a subobject p'' which is not a base of p. If its dynamic type is, say, p', then, necessarily, p' is in state BasesConstructed or StartedDestructing (as the most-derived object cannot be Constructed). By unicity, $p' = p$, which is absurd.

□

Conversely:

Lemma 28.

- *If the most-derived object becomes Constructed, then nothing happens on the dynamic types.*

- *Otherwise, if a subobject p becomes other than* BasesConstructed *or* StartedDestructing*, then the dynamic type of an object cannot change to a defined value.*
- *Otherwise, if no subobject changes its construction state, then no dynamic type changes.*

Proof.

- In the first case, the construction state of the most-derived object passes from BasesConstructed to Constructed. So, in both cases, the dynamic type of all bases has already switched to the most-derived object.
- In the second case, consider a subobject p'' whose dynamic type becomes defined as a subobject p' in state BasesConstructed or StartedDestructing. Then, $p' \neq p$ (as p is no longer in such a construction state). So, p' was not affected by the construction state change, so it was already BasesConstructed or StartedDestructing before the construction state change. So, the dynamic type of p'' was already p', so it has not changed.
- The third case is trivial.

□

Those two theorems point out the precises times when a compiler implementation has to actually change the dynamic type data to reflect the dynamic type at the implementation level: when all bases are constructed, and just before the construction of fields, the pointers to virtual tables change for the subobject and all of its bases, as well as entering the destructor.

8 Conclusion and future work

To the best of our knowledge, the major novelty of our work is a formal account on the metatheory of C++ object construction and destruction for the whole C++ multiple inheritance object model, including some RAII properties.
Our formalization led to proposed changes in the C++ Standard (most notably, CWG 1202 about virtual functions during the destruction of the fields of a class, has just been integrated into C++0x, as voted in March 2011).
Our work points out the precise execution points when the generalized dynamic type of an object changes. This information can help build a verified compiler from our language to an intermediate language with explicit concrete virtual tables. (Ongoing work.)
Restrictions on the semantics still keep a relatively realistic language, cf. Lockheed Martin and Embedded C++. However, one may consider extending the formal semantics to include free store, temporaries, functions returning structures.