Formal Verification of C++
Object Construction and Destruction

Tahina Ramananandro\textsuperscript{1}

\textsuperscript{1}INRIA Paris-Rocquencourt

November 18th, 2011
Outline

1. Introduction
2. Formal semantics of C++ object model
3. Object construction and destruction
4. Application to Verified compilation
5. Conclusion and perspectives
Outline

1 Introduction
   - Construction: object initialization
   - Destruction: resource management
   - A brief overview of C++ multiple inheritance
   - Overview of our work

2 Formal semantics of C++ object model

3 Object construction and destruction

4 Application to Verified compilation

5 Conclusion and perspectives
Initializing objects

```cpp
struct Point {
    double x;
    double y;
};
```
Initializing objects

```c
struct Point {
    double x;
    double y;
};

main () {
    Point c;
    c.x = 1.2;
    c.y = 3.4;
}
```
Initializing objects

```cpp
struct Point {
  double x;
  double y;
};

main () {
  Point c = {1.2, 3.4};
}
```
Initializing objects using a constructor

```cpp
struct Point {
    double x;
    double y;
    Point (double x0, double y0) {
        x = x0;
        y = y0;
    }
};

main () {
    Point c = Point (1.2, 3.4);
}
```
Initializing objects using a constructor

```cpp
struct Point {
    double x;
    double y;
    Point (double x0, double y0): x(x0), y(y0) {}
};

main () {
    Point c = Point (1.2, 3.4);
}
```
Initializing embedded objects

```c
struct Segment {
    Point p1;
    Point p2;
    Segment (double x1, double y1, double x2, double y2):
        p1 (x1, y1), p2 (x2, y2) {}
}
main () {
    Segment s = Segment (1.2, 3.4, 18.42, 17.29);
}
```
Initializing inherited subobjects

```
struct ColoredPoint: Point {
    int color;
    ColoredPoint (double x0, double y0, int color0):
        Point (x0, y0), color(color0) {}
}
main () {
    ColoredPoint c = ColoredPoint (1.2, 3.4, 256);
}
```
Outline

1 Introduction
   - Construction: object initialization
   - Destruction: resource management
   - A brief overview of C++ multiple inheritance
   - Overview of our work

2 Formal semantics of C++ object model

3 Object construction and destruction

4 Application to Verified compilation

5 Conclusion and perspectives
main () {
    File f = File ("toto.txt");
    f.write ("Hello world!");
}
Object destruction

```c
struct File {
    FILE* handle;
    File (char* name): handle (fopen (name, "w")) {}
    virtual void write (char* string) {
        fputs (handle, string);
    }
    ~File () {
        fclose (handle);
    }
}

main () {
    File f = File ("toto.txt");
    f.write ("Hello world!");
}
```
Destructing embedded objects

```cpp
struct LockFile {
    Lock lock;
    File file;
    LockFile (char* name): lock (), file (name) {}
};

Two subobjects of the same object must be destructed in the reverse order of their destruction.
```
Destructing inherited objects

Java and C# are buggy:

class File implements Closeable {
    public void close () {...}
}
class BuggyFile extends File {
    public void close () {}
}

try (File f = new BuggyFile("toto.txt")) {
    ...
}

File is not closed properly. By contrast, C++ guarantees that destructors for base classes are called.
Focus of our work

A study of object construction and destruction for C++ objects.
Outline

1 Introduction
   - Construction: object initialization
   - Destruction: resource management
   - A brief overview of C++ multiple inheritance
   - Overview of our work

2 Formal semantics of C++ object model

3 Object construction and destruction

4 Application to Verified compilation

5 Conclusion and perspectives
Single inheritance

```cpp
struct PluggedDevice {
    int plug;
};

struct Component: PluggedDevice {
    int switch;
};

struct Clock: Component {}

struct Radio: Component {
    int volume;
}
```
Two kinds of multiple inheritance

struct PluggedDevice {
    int plug;
}

struct Component : 
    virtual PluggedDevice {
    int switch;
}

struct Clock: Component {
    int time;
}

struct Radio: Component {
    int volume;
}

struct Alarm: Clock, Radio {
    int alarmTime;
}
The algebra of subobjects

Previous works:
- Rossie & Friedman (OOPSLA’95)
- Wasserrab, Nipkow & al. (OOPSLA’06)

Path from the full class or a virtual base, to the dynamic type of the pointer, only through non-virtual inheritance.

If $D$ derives from $B$, then every virtual base of $D$ is a virtual base of $B$. 
The algebra of subobjects

From Alarm to Component:
- Alarm :: Clock :: Component :: nil
- Alarm :: Radio :: Component :: nil
- Alarm :: Component :: nil

From Alarm to PluggedDevice:
- PluggedDevice :: nil
Outline

1. Introduction
   - Construction: object initialization
   - Destruction: resource management
   - A brief overview of C++ multiple inheritance
   - Overview of our work

2. Formal semantics of C++ object model

3. Object construction and destruction

4. Application to Verified compilation

5. Conclusion and perspectives
Overview of our work

- A formalization of the semantics of C++ objects, with the main interesting features:
  - multiple inheritance
  - virtual inheritance
  - embedded structure fields
  - static and dynamic casts, virtual function calls
  - object construction and destruction

- Properties of object construction and destruction
- A verified compiler to a Cminor-style 3-address language with low-level memory accesses
- All proofs done with the Coq proof assistant
Outline

1. Introduction
2. Formal semantics of C++ object model
3. Object construction and destruction
4. Application to Verified compilation
5. Conclusion and perspectives
History of formal semantics of C++ subobjects

- First formalization: Rossie & Friedman, *An algebraic semantics of subobjects* (OOPSLA’95)
- First machine formalization: Wasserrab, Nipkow et al., *An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++* (OOPSLA’06)
Designating subobjects with paths

\[ n_{D,B} ::= D :: \cdots :: B \]

Non-virtual inheritance path

\[ p_{D,B} ::= (\text{Repeated}, n_{D,B}) \]

\[ (\text{Shared}, n_{V,B}) \]

\( B \) is a non-virtual base of \( D \)

\( V \) is a virtual base of \( D \)

and \( B \) is a non-virtual base of \( V \)
Designating subobjects with paths

We extended those works to embedded structures and arrays.

\[ n_{D,B} ::= D :: \cdots :: B \]  
Non-virtual inheritance path

\[ p_{D,B} ::= (\text{Repeated}, n_{D,B}) \] \( B \) is a non-virtual base of \( D \)

\[ | (\text{Shared}, n_{V,B}) \] \( V \) is a virtual base of \( D \) and \( B \) is a non-virtual base of \( V \)

\[ \text{subo} ::= (idx, p, f)^* (idx', p') \] path to a subobject inside an array through embedded structure array fields
A core language

We defined a core language for C++ multiple inheritance, featuring the most interesting object-oriented features:

\[ Stmt ::= \begin{align*}
& \text{var} := \text{var} \rightarrow_C \text{fc} & \text{Reading scalar field or pointing to structure field} \\
& \text{var} \rightarrow_C \text{fc} := \text{var} & \text{Writing scalar field} \\
& \text{var} := \& \text{var}[\text{var}]_C & \text{Pointing to array cell} \\
& \text{var} := \text{static\_cast}^{\langle A \rangle}_C(\text{var}) & \text{Static cast} \\
& \text{var} := \text{dynamic\_cast}^{\langle A \rangle}_C(\text{var}) & \text{Dynamic cast} \\
& \text{var} := \text{var} \rightarrow_C \text{fc}(\text{var}, \ldots) & \text{Virtual function call} \\
& \{ \text{Cc}[n] = \{ \text{Init}_C, \ldots \}; \text{Stmt} \} & \text{Block-scoped object} \\
& \ldots & \text{Structured control} \\
\end{align*} \]

\[ Init_C ::= \text{Stmt}; C(\text{var}, \ldots) \]

Initializer
A core language

We defined a core language for C++ multiple inheritance, featuring the most interesting object-oriented features:

\[
\begin{align*}
Funct & ::= \text{virtual } f(var, \ldots)\{Stmt\} & \text{Virtual function definition} \\
Finit_m & ::= m\{Init_A\} & \text{Structure data member initializer for } A m[n] \\
| & m(Stmt, var) & \text{Scalar data member initializer} \\
Constr_C & ::= C(var, \ldots) : Init_{B1}, \ldots, Init_{V1}, \ldots, \\
& \quad Finit_m, \ldots \{Stmt\} & \text{Constructor} \\
Class & ::= \text{struct } C : B1, \ldots, \text{virtual } V1, \ldots \{ \\
& \quad Constr_C, \ldots \\
& \quad Funct, \ldots \\
& \} & \text{Class definitions}
\end{align*}
\]
Outline

1. Introduction
2. Formal semantics of C++ object model
3. Object construction and destruction
4. Application to Verified compilation
5. Conclusion and perspectives
The semantics of object construction and destruction

We have designed a small-step operational semantics precisely modeling the different steps of object construction and destruction. The semantics has to tackle the following two issues:

- In which order are subobjects constructed and destructed?
- Which virtual functions are called within a constructor?
The construction states of a subobject

Each (inheritance and/or embedded structure) subobject is equipped at run-time with a *construction state*:

```
Unconstructed          Destructed
  ↓                    ↓
StartedConstructing   DestructingBases
  ↓                    ↓
BasesConstructed      StartedDestructing
  ↓
Constructed
```

The *lifetime* of a subobject is the set of all states where the construction state of the object is Constructed.
Evolution of the construction state during construction

```cpp
struct C : B {
    int i;
    C() : B(), i(18) { ... }
}

Unconstructed
```
Evolution of the construction state during construction

```cpp
struct C : B {
    int i;
    C() : B(), i(18) {}
}
```

StartedConstructing
Evolution of the construction state during construction

```cpp
struct C : B {
    int i;
    C () : B (), i(18) { ... }
}

BasesConstructed, virtual functions allowed here
```
Evolution of the construction state during construction

```cpp
struct C: B {
  int i;
  C(): B(), i(18) {...}
}
```

Constructed
Run-time invariant

To reason about the semantics, we have to specify and prove a run-time invariant. (13000 kloc, 2 hours checking time)
Lemma

If $p$ is a direct subobject of $p'$:

- direct non-virtual base subobject
- direct or indirect virtual base (if $p'$ is a most-derived object)
- array cell of a structure field

Then the following table relates their construction states:

<table>
<thead>
<tr>
<th>If $p'$ is...</th>
<th>Then $p$ is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconstructed</td>
<td>Unconstructed</td>
</tr>
<tr>
<td>StartedConstructing</td>
<td>Unconstructed if $p$ is a field subobject of $p'$ between Unconstructed and Constructed otherwise</td>
</tr>
<tr>
<td>BasesConstructed</td>
<td>Constructed if $p$ is a base subobject of $p'$ between Unconstructed and Constructed otherwise</td>
</tr>
<tr>
<td>Constructed</td>
<td>Constructed</td>
</tr>
<tr>
<td>StartedDestructing</td>
<td>Constructed if $p$ is a base subobject of $p'$ between Constructed and Destructed otherwise</td>
</tr>
<tr>
<td>DestructingBases</td>
<td>Destructed if $p$ is a field subobject of $p'$ between Constructed and Destructed otherwise</td>
</tr>
<tr>
<td>Destructed</td>
<td>Destructed</td>
</tr>
</tbody>
</table>
Lemma

Let $p_1, p_2$ two sibling subobjects such that $p_1$ appears before $p_2$ in the construction tree. Then, the following table relates their construction states:

<table>
<thead>
<tr>
<th>If $p_1$ is...</th>
<th>Then $p_2$ is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconstructed</td>
<td>Unconstructed</td>
</tr>
<tr>
<td>StartedConstructing</td>
<td>Unconstructed</td>
</tr>
<tr>
<td>BasesConstructed</td>
<td>in an arbitrary state</td>
</tr>
<tr>
<td>Constructed</td>
<td>Destructed</td>
</tr>
<tr>
<td>StartedDestructing</td>
<td>Destructed</td>
</tr>
<tr>
<td>DestructingBases</td>
<td>Destructed</td>
</tr>
<tr>
<td>Destructed</td>
<td></td>
</tr>
</tbody>
</table>
Theorem

Each object is constructed and destructed exactly once, in this order.

Theorem

If an object is constructed, then all its subobjects are constructed.

Theorem

If an object is deallocated, then it and all its subobjects are previously constructed, then destructed, in this order.

Theorem

Two subobjects of the same allocated object are destructed in the reverse order of their construction.
The generalized dynamic type of a subobject

A subobject $\sigma$ has a generalized dynamic type $\sigma_\circ$ if, and only if:

- either $\sigma_\circ$ is the most-derived object, and it is Constructed (i.e. whole construction has ended and destruction has not started yet)
- or $\sigma_\circ$ is BasesConstructed or StartedDestructing and $\sigma$ is an inheritance subobject of $\sigma_\circ$

$\sigma_\circ$ is then considered as the most-derived object for polymorphic operations (dynamic cast, virtual function call). In practice, $\sigma_\circ$ corresponds to the object whose body of constructor/destructor is running.
Thick transitions show the times when the compiler must update the pointers to virtual tables.
Outline

1. Introduction
2. Formal semantics of C++ object model
3. Object construction and destruction
4. Application to Verified compilation
5. Conclusion and perspectives
Outline

1. Introduction

2. Formal semantics of C++ object model

3. Object construction and destruction

4. Application to Verified compilation
   - Compiling core C++ object-oriented features
   - Compiling object constructors and destructors
   - Semantics preservation
   - A brief overview of C++ object layout

5. Conclusion and perspectives
Compilation of object-oriented operations

\[ [x := x' \rightarrow_C F] = x := \text{load}(\text{scsize}_t, x' + \text{foff}_C(F)) \]
\[(\text{if } F = (f, t) \text{ is a scalar field of } C)\]

\[ [x \rightarrow_C F := x'] = \text{store}(\text{scsize}_t, x + \text{foff}_C(F), x') \]
\[(\text{if } F = (f, t) \text{ is a scalar field of } C)\]

\[ [x := x' \rightarrow_C F] = x := x' + \text{foff}_C(F) \]
\[(\text{if } F \text{ is a structure array field of } C)\]

\[ [x := \&x_1[x_2]_C] = x := x_1 + \text{size}_C \times x_2 \]

\[ [x := x_1 == x_2] = x := x_1 == x_2 \]
Compilation of casts

- For static casts, there are two cases:
  - For a non-virtual subobject $p_{D,B} = (\text{Repeated, } I)$:
    
    \[
    \begin{align*}
    [x := \text{static\_cast}\langle B\rangle_D(x')] &= x := x' + \text{nvsoff}(I) \\
    [x := \text{static\_cast}\langle D\rangle_B(x')] &= x := x' - \text{nvsoff}(I)
    \end{align*}
    \]

  - For a subobject through virtual inheritance $p_{D,B} = (\text{Shared, } V :: I)$, the offset of the virtual base $V$ of $C$ must be looked up in the dynamic type data:
    
    \[
    \begin{align*}
    [x := \text{static\_cast}\langle A\rangle_C(x')] &= \\
    t := \text{load}(\text{dtdatasize}, x'); x := x' + \text{read\_vboff}(t, V) + \text{nvsoff}(I)
    \end{align*}
    \]
    
    (reads through dynamic type data are left abstract)
Compilation of casts

- For static casts, there are two cases:
  - For a non-virtual subobject \( p_{D,B} = \text{(Repeated,} \, l) \):
    
    \[
    \begin{align*}
    [x := \text{static
d_cast} \langle B \rangle_D(x')] &= x := x' + \text{nvsoff}(l) \\
    [x := \text{static
d_cast} \langle D \rangle_B(x')] &= x := x' - \text{nvsoff}(l)
    \end{align*}
    \]

  - For a subobject through virtual inheritance \( p_{D,B} = \text{(Shared,} \, V :: \, l) \), the offset of the virtual base \( V \) of \( C \) must be looked up in the dynamic type data:
    
    \[
    \begin{align*}
    [x := \text{static
d_cast} \langle A \rangle_C(x')] &= \\
    t := \text{load(dtdatasize,} x'); x := x' + \text{read_vboff}(t, V) + \text{nvsoff}(l)
    \end{align*}
    \]

    (reads through dynamic type data are left abstract)

- Dynamic cast is compiled as a read through the pointer to dynamic type data
Outline

1. Introduction

2. Formal semantics of C++ object model

3. Object construction and destruction

4. Application to Verified compilation
   - Compiling core C++ object-oriented features
   - Compiling object constructors and destructors
   - Semantics preservation
   - A brief overview of C++ object layout

5. Conclusion and perspectives
void _constr_C(bool isMostDerived, C* this, ...) {
  if(isMostDerived) {
    for each V direct or indirect virtual base of C {
      execute the initializer for V, ending with
      _constr_V(false, (V*) this, ...);
    }
  }
  for each B direct non-virtual base of C {
    execute the initializer for B, ending with
    _constr_B(false, (B*) this, ...);
  }
  set dynamic type to C;
  for each m data member of C {
    if m is a scalar {
      execute the initializer for m, ending with
      this->m = value;
    } else, m is a structure A[n] {
      for(i = 0, i < n, ++i) {
        execute the initializer for m[i], ending with
        _constr_A(true, &(this->m[i]), ...);
      }
    }
  }
};
execute the constructor body;
return;
Outline

1. Introduction

2. Formal semantics of C++ object model

3. Object construction and destruction

4. Application to Verified compilation
   - Compiling core C++ object-oriented features
   - Compiling object constructors and destructors
   - Semantics preservation
   - A brief overview of C++ object layout

5. Conclusion and perspectives
Semantics preservation

Theorem

The compilation scheme preserves the semantics of programs through forward simulation:

\[ \mathcal{J} \ni s \circ \triangledown s_1 \Rightarrow s_2 \ni \mathcal{F} z \]

\[ \mathcal{J}' \ni s_0 \circ \star \Rightarrow s_1 \]

\[ s_1 \ni \epsilon? \Rightarrow s_2 \ni \mathcal{F} z \]

\[ s_0' \ni \star \Rightarrow s_f' \ni \mathcal{F} z' \]
Theorem

The compilation scheme preserves the semantics of programs through forward simulation:

\[ \mathcal{I} \ni s \xrightarrow{\epsilon} \mathcal{F}_z \]

\[ \mathcal{I}' \ni s'_o \xrightarrow{\epsilon^?} s'_1 \]

\[ s_1 \xrightarrow{\epsilon^?} s'_1 \xrightarrow{\epsilon^?} s'_2 \]

\[ s_f \in \mathcal{F}'_z \]
Outline

1. Introduction

2. Formal semantics of C++ object model

3. Object construction and destruction

4. Application to Verified compilation
   - Compiling core C++ object-oriented features
   - Compiling object constructors and destructors
   - Semantics preservation
   - A brief overview of C++ object layout

5. Conclusion and perspectives
C++ multiple inheritance issues on data layout

Usual layout problems:
- alignment padding
- embedded structures: possibility of reusing padding?
C++ multiple inheritance issues on data layout

Usual layout problems:
- alignment padding
- embedded structures: possibility of reusing padding?

Issues raised by multiple inheritance:
- Dynamic type data (e.g. pointers to virtual tables)
  - needed for dynamic cast, virtual function dispatch
  - even field accesses through virtual inheritance
  - not ordinary fields, may be shared between subobjects
- Object identity: two pointers to different subobjects of the same type must compare different, even in the presence of empty bases.
Common vendor ABI layout algorithm

- Application Binary Interface: agreement on data layout for programs compiled by different compilers for the same platform
- Initially for Itanium, then adopted by GNU GCC and almost all compiler builders and platforms (except Microsoft)
- A fairly complicated algorithm, difficult to implement
Common vendor ABI layout algorithm

Chapter 2: Data Layout

2.1 General

In what follows, we define the memory layout for C++ data objects. Specifically, for each type, we specify the following information about an object O of that type:

- the size of an object, sizeof(O);
- the alignment of an object, align(O); and
- the offset within O, offset(O), of each data component C, i.e. base or member.

For purposes internal to the specification, we also specify:

- dsize(O): the data size of an object, which is the size of O without tail padding.
- nvsize(O): the non-virtual size of an object, which is the size of O without virtual bases.
- nvalign(O): the non-virtual alignment of an object, which is the alignment of O without virtual bases.

2.2 POD Data Types

The size and alignment of a type which is a POD for the purpose of layout is as specified by the base (C) ABI. Type bool has size and alignment 1. All of these types have data size and non-virtual size equal to their size. (We ignore tail padding for PODs because the Standard does not allow us to use it for anything else.)

2.3 Member Pointers

A pointer to data member is an offset from the base address of the class object containing it, represented as a ptrdiff_t. It has the size and alignment attributes of a ptrdiff_t. A NULL pointer is represented as -1.

A pointer to member function is a pair as follows:

ptr: For a non-virtual function, this field is a simple function pointer. (Under current base Itanium psABI conventions, that is a pointer to a GP/function address pair.) For a virtual function, it is 1 plus the virtual table offset (in bytes) of the function, represented as a ptrdiff_t. The value zero represents a NULL pointer, independent of the adjustment field value below.

adj: The required adjustment to this, represented as a ptrdiff_t.
It has the size, data size, and alignment of a class containing those two members, in that order. (For 64-bit Itanium, that will be 16, 16, and 8 bytes respectively.)

2.4 Non-POD Class Types

For a class type C which is not a POD for the purpose of layout, assume that all component types (i.e. proper base classes and non-static data member types) have been laid out, defining size, data size, non-virtual size, alignment, and non-virtual alignment. (See the description of these terms in General above.) Further, assume

Correctness of the common vendor ABI layout algorithm

Theorem

This algorithm can be fed to the compiler to obtain a verified compiler preserving the semantics of programs.

Object layout entirely proved except a controversial optimization on virtual primary bases.

We developed and proved the correctness of an extension of this algorithm to allow further reusing of the tail paddings of non-virtual bases and fields.
Outline

1. Introduction
2. Formal semantics of C++ object model
3. Object construction and destruction
4. Application to Verified compilation
5. Conclusion and perspectives
Summary

- A general formal model for C++ object-oriented features
- First machine-checked formalization of RAI
- First machine-checked correctness proof of verified compiler for C++ object construction and destruction
- Positive feedback from C++ Standard Committee: some standard issues corrected, some other pending

Quite a long formalization (80 kloc, 3 hours checking time), but the semantics itself is tractable (900 lines).
Future work

Extending the semantics:

- Free store
- C++ copy semantics (passing constructor arguments by value, copy constructor, functions returning structures)
- Exceptions? (Excluded by Lockheed Martin)
- Templates (Sieck et al., ECOOP’06)

Improving the compiler:

- Concrete representation of virtual tables and VTT
- Virtual primary bases
- Better object layout algorithms (bidirectional, etc.)
Thank you for your attention

- Coq development fully available on the Web:
  http://gallium.inria.fr/~tramanan/cxx/compiler
- For further information: Tahina.Ramananandro@inria.fr
Virtual primary bases

```c
struct A {
    virtual void f();
};
struct V : virtual A
struct C : virtual A
struct B1 : virtual V
struct B2 : virtual V
struct D : C, B1, B2
```
Thank you for your attention

Tahina.Ramananandro@inria.fr
http://gallium.inria.fr/~tramanan/cxx/object-layout

1. Introduction
   - Construction: object initialization
   - Destruction: resource management
   - A brief overview of C++ multiple inheritance
   - Overview of our work

2. Formal semantics of C++ object model

3. Object construction and destruction

4. Application to Verified compilation
   - Compiling core C++ object-oriented features
   - Compiling object constructors and destructors
   - Semantics preservation
   - A brief overview of C++ object layout

5. Conclusion and perspectives
   - Virtual primary bases
   - Thank you