Formal Verification of C++ Object Construction and Destruction

Tahina Ramananandro[1]

[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

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

Initializing objects

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

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

Initializing objects

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

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

Initializing objects using a constructor

```
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

```
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

```
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

Object destruction

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

Object destruction

```
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

```
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.

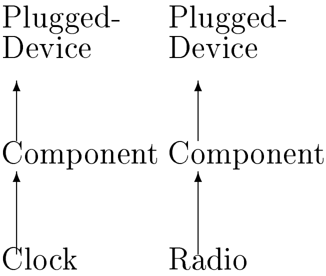
Focus of our work

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

Outline

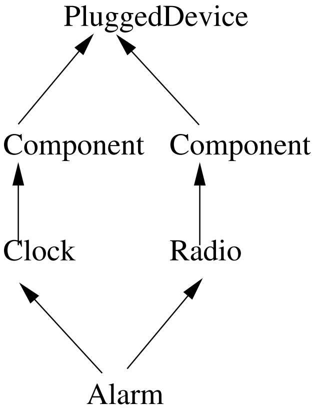
1. Introduction
   - Construction: object initialization
   - Destruction: resource management
   - A brief overview of C++ multiple inheritance
   - Overview of our work
2. Formal semantics of C++ object model
3. Object construction and destruction
4. Application to Verified compilation
5. Conclusion and perspectives

Single inheritance



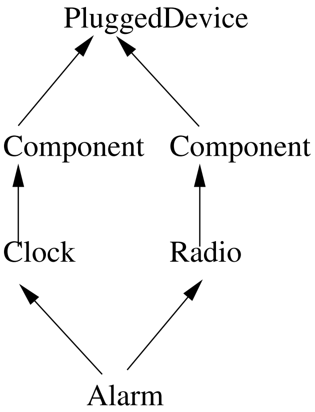
```
struct PluggedDevice {
  int plug;
}

struct Component: PluggedDevice {
  int switch;
}

struct Clock: Component {}

struct Radio: Component {
  int volume;
}
```

Two kinds of multiple inheritance



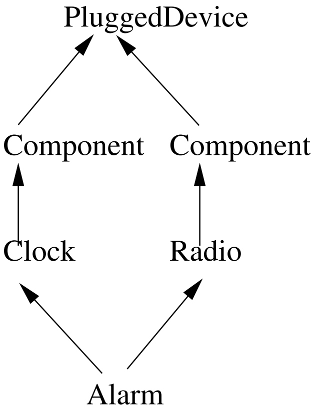
```
struct PluggedDevice {
  int plug;
}

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

$$
\begin{array}{rcll}
nv_{D,B} & ::= & D :: \cdots :: B & \text{Non-virtual inheritance path} \\
p_{D,B} & ::= & (\text{Repeated}, nv_{D,B}) & B \text{ is a non-virtual base of } D \\
 & \mid & (\text{Shared}, nv_{V,B}) & V \text{ is a virtual base of } D \\
 & & & \text{and } B \text{ is a non-virtual base of } V
\end{array}
$$

Designating subobjects with paths

We extended those works to embedded structures and arrays.

$$\begin{array}{rcll} nv_{D,B} & ::= & D :: \cdots :: B & \text{Non-virtual inheritance path} \\ p_{D,B} & ::= & (\mathrm{Repeated}, nv_{D,B}) & B \text{ is a non-virtual base of } D \\ & | & (\mathrm{Shared}, nv_{V,B}) & V \text{ is a virtual base of } D \\ & & & \text{and } B \text{ is a non-virtual base of } V \\ subo & ::= & (idx, p, f)^* \ (idx', p') & \text{path to a subobject inside an array} \\ & & & \text{through embedded structure array fields} \end{array}$$

A core language

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

$Stmt$	$::=$	$var := var\texttt{->}_C f$	Reading scalar field or pointing to structure field
	$\mid$	$var\texttt{->}_C f := var$	Writing scalar field
	$\mid$	$var := \&var[var]_C$	Pointing to array cell
	$\mid$	$var := \texttt{static\_cast}\langle A \rangle_C(var)$	Static cast
	$\mid$	$var := \texttt{dynamic\_cast}\langle A \rangle_C(var)$	Dynamic cast
	$\mid$	$var := var\texttt{->}_C f(var, \dots)$	Virtual function call
	$\mid$	$\{Cc[n] = \{Init_C, \dots\}; Stmt\}$	Block-scoped object
	$\mid$	$\dots$	Structured control
$Init_C$	$::=$	$Stmt; C(var, \dots)$	Initializer

A core language

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

$Funct$	$::=$	$\texttt{virtual}\ f(var, \ldots)\{Stmt\}$	Virtual function defi
$Finit_m$	$::=$	$m\{Init_A\}$	Structure data mem for $A\ m[n]$
	$\mid$	$m(Stmt, var)$	Scalar data member
$Constr_C$	$::=$	$C(var, \ldots) : Init_{B1}, \ldots, Init_{V1}, \ldots,$ $Finit_m, \ldots \{Stmt\}$	Constructor
$Class$	$::=$	$\texttt{struct}\ C : B1, \ldots, \texttt{virtual}\ V1, \ldots \{$ $Constr_C, \ldots$ $Funct, \ldots$ $\}$	Class definitions

Outline

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

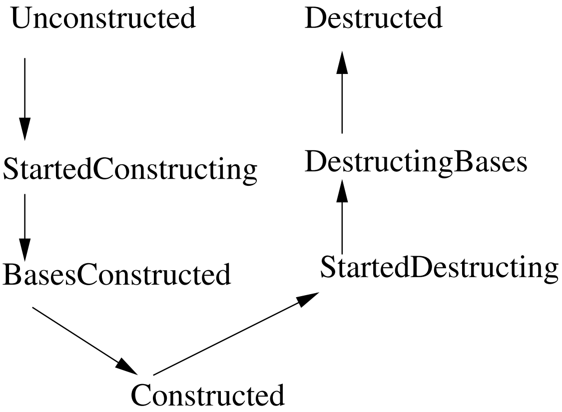
The semantics of object construction and destruction

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*:



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

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

Unconstructed

Evolution of the construction state during construction

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

StartedConstructing

Evolution of the construction state during construction

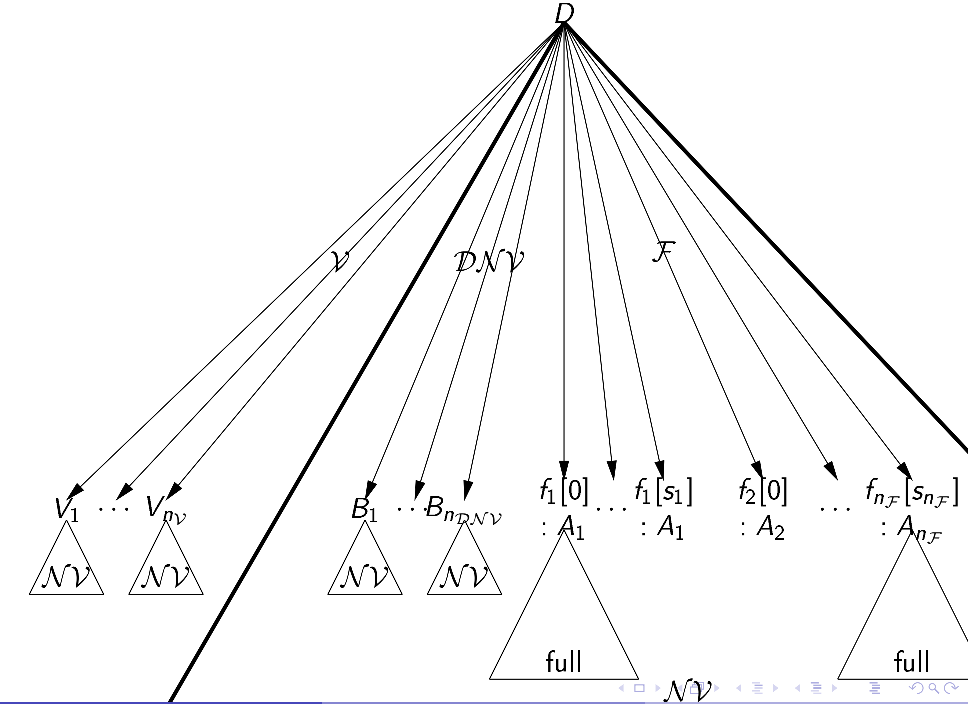
```
struct C: B {
  int i;
  C (): B (), i(18) {...}
}
```

BasesConstructed, virtual functions allowed here

Evolution of the construction state during construction

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

Constructed


D
$\mathcal{V}$
$\mathcal{DNV}$
$\mathcal{F}$
V_1 … $V_{n_\mathcal{V}}$
B_1 … $B_{n_\mathcal{DNV}}$
$f_1[0]$: A_1
…
$f_1[s_1]$: A_1
$f_2[0]$: A_2
…
$f_{n_\mathcal{F}}[s_{n_\mathcal{F}}]$: $A_{n_\mathcal{F}}$
$\mathcal{NV}$
$\mathcal{NV}$
$\mathcal{NV}$
$\mathcal{NV}$
full
full
$\mathcal{NV}$

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:

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

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:

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

RAII

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

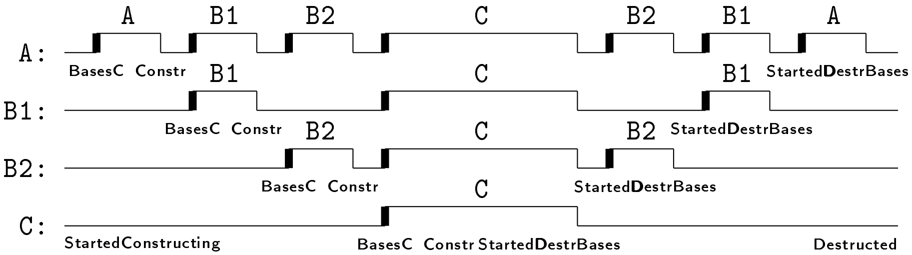
Two subobjects of the same allocated object are destructed in the reverse order of their construction.

The generalized dynamic type of a subobject

A subobject σ 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 σ 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

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

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

$$[\![x := x'\texttt{->}_C F]\!] = x := x' + \mathrm{foff}_C(F)$$
$$\text{(if } F \text{ is a structure array field of } C\text{)}$$

$$[\![x := \&x_1[x_2]_C]\!] = x := x_1 + \mathrm{size}_C \times x_2$$

$$[\![x := x_1 \texttt{ == } x_2]\!] = x := x_1 \texttt{ == } x_2$$

Compilation of casts

- For static casts, there are two cases:
  - For a non-virtual subobject $p_{D,B} = (\text{Repeated}, l)$:

$$\llbracket x := \texttt{static\_cast}\langle B\rangle_D(x')\rrbracket = x := x' + \text{nvsoff}(l)$$
$$\llbracket x := \texttt{static\_cast}\langle D\rangle_B(x')\rrbracket = x := x' - \text{nvsoff}(l)$$

  - 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:

$$\llbracket x := \texttt{static\_cast}\langle A\rangle_C(x')\rrbracket = \\ t := \texttt{load}(\text{dtdatasize}, x'); x := x' + \texttt{read\_vboff}(t, V) + \text{nvsoff}(l)$$

(reads through dynamic type data are left abstract)

Compilation of casts

- For static casts, there are two cases:
  - For a non-virtual subobject $p_{D,B} = (\text{Repeated}, l)$:

$$\llbracket x := \texttt{static\_cast}\langle B \rangle_D (x') \rrbracket = x := x' + \text{nvsoff}(l)$$
$$\llbracket x := \texttt{static\_cast}\langle D \rangle_B (x') \rrbracket = x := x' - \text{nvsoff}(l)$$

  - 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:

$$\llbracket x := \texttt{static\_cast}\langle A \rangle_C (x') \rrbracket = \\ t := \texttt{load}(\text{dtdatasize}, x'); x := x' + \texttt{read\_vboff}(t, V) + \text{nvsoff}(l)$$

  (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:

$$\begin{tikzcd}
\mathfrak{I} \arrow[rr, no head, "\ni"] & & s_\circ \arrow[d, dotted, no head, "\triangleright"] \\
\mathfrak{I}' \arrow[r, dotted, no head, "\ni"] & s'_\circ \arrow[r, dotted, "'\star"] & s'_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s_1 \arrow[r, "\mathfrak{e}^?"'] \arrow[d, no head, "\triangleright"] & s_2 \arrow[d, dotted, no head, "\triangleright"] \\
s'_1 \arrow[r, dotted, "'+", "\mathfrak{e}^?"'] & s'_2
\end{tikzcd}
\qquad
\begin{tikzcd}
s_f \arrow[rr, no head, "\in"] \arrow[d, no head, "\triangleright"] & & \mathfrak{F}_z \\
s'_{99} \arrow[r, dotted, "'\star"] & s'_f \arrow[r, dotted, no head, "\in"] & \mathfrak{F}'_z
\end{tikzcd}$$

Semantics preservation

Theorem

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

Proved in Coq

$$\begin{tikzcd}
\mathfrak{I} \arrow[rr, dash, "\ni"] & & s_\circ \arrow[d, dotted, "\triangleright"] \\
\mathfrak{I}' \arrow[r, dotted, dash, "\ni"] & s'_\circ \arrow[r, dotted, "'\star"] & s'_1
\end{tikzcd}
\qquad
\begin{tikzcd}
s_1 \arrow[r, "\mathfrak{e}^?"'] \arrow[d, dash, "\triangleright"] & s_2 \arrow[d, dotted, dash, "\triangleright"] \\
s'_1 \arrow[r, dotted, "'+", "\mathfrak{e}^?"'] & s'_2
\end{tikzcd}
\qquad
\begin{tikzcd}
s_f \arrow[rr, dash, "\in"] \arrow[d, dash, "\triangleright"] & & \mathfrak{F}_z \\
s'_{99} \arrow[r, dotted, "'\star"] & s'_f \arrow[r, dotted, dash, "\in"] & \mathfrak{F}'_z
\end{tikzcd}$$

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?

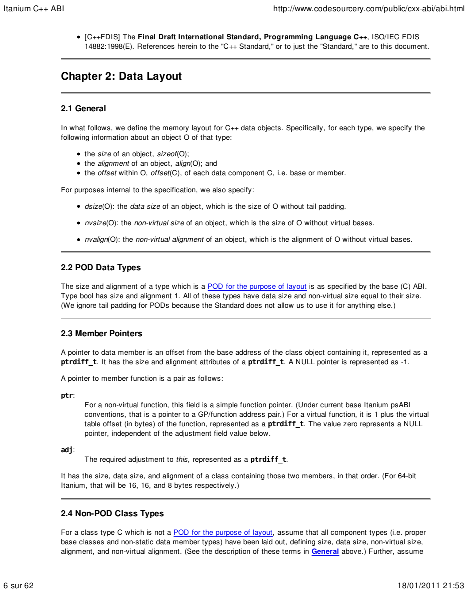
Issues raised by multiple inheritance:

- 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
- Common vendor ABI designed by a consortium of compiler designers, `http://www.codesourcery.com/public/cxx-abi/`
- 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



Itanium C++ ABI http://www.codesourcery.com/public/cxx-abi/abi.html

- [C++FDIS] The **Final Draft International Standard, Programming Language C++**, ISO/IEC FDIS 14882:1998(E). References herein to the "C++ Standard," or to just the "Standard," are to this document.

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*(C), 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`:

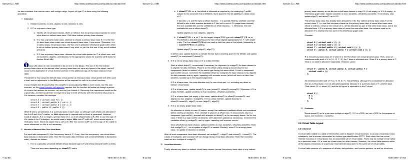
The required adjustment to *this*, represented as a `ptrdiff_t`.

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

6 sur 62 18/01/2011 21:53



Correctness of the common vendor ABI layout algorithm

Theorem

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

Proved in Coq

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 RAII
- 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 (Siek et al., ECOOP'06)

Improving the compiler:

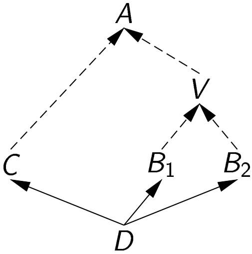
- Concrete representation of virtual tables and VTT
- Virtual primary bases
- Better object layout algorithms (bidirectional, etc.)

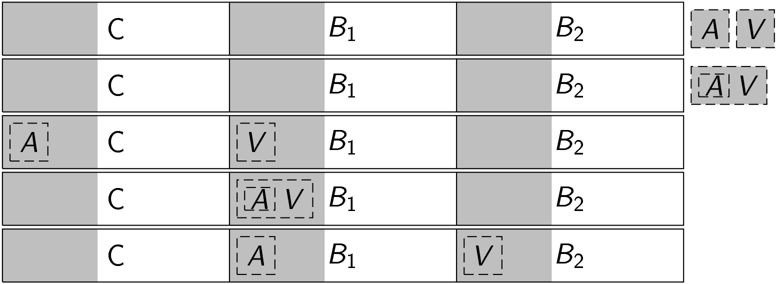
Thank you for your attention

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

Virtual primary bases

struct A { `virtual void` f(); };
struct V : `virtual` A
struct C : `virtual` A
struct B_1 : `virtual` V
struct B_2 : `virtual` V
struct D : C, B_1, B_2





Thank you for your attention

`Tahina.Ramananandro@inria.fr`

`http://gallium.inria.fr/~tramanan/cxx/object-layout`