A machine-checked formalization of concrete object layout for C++ multiple inheritance

Tahina Ramananandro[1] Xavier Leroy[1]

[1] Gallium Team-project
INRIA Paris-Rocquencourt

February 17th, 2010

Photo courtesy of François Pottier

Motivation

Many formal methods and tools exist to make program analysis easier for object-oriented languages (JML, Jahob, Krakatoa, Spec♯, ...). **But :**

- most based upon Java/C♯, only allow single inheritance and interfaces
- all restricted to the source-code level

Motivation

- Our goal : link formal presentation of multiple inheritance with an actual, realistic low-level implementation

Motivation

- Our goal : link formal presentation of multiple inheritance with an actual, realistic low-level implementation
- Our case study : C++ multiple inheritance
  - combines two different schemes of multiple inheritance
  - must take care of performance
  - widely used... but often avoided for safety-critical software because of perceived complexity

Motivation

- Our goal : link formal presentation of multiple inheritance with an actual, realistic low-level implementation
- Our case study : C++ multiple inheritance
  - combines two different schemes of multiple inheritance
  - must take care of performance
  - widely used... but often avoided for safety-critical software because of perceived complexity
- Our companion : The Coq proof assistant !
  - a language and software to write mathematical specifications and mechanically prove theorems about those specifications. (Other examples are ACL2, Isabelle/HOL, etc.)
  - Proofs are not automatic, but interactive and the prover rechecks the proof input by the user.
  - Coq also allows to extract trustworthy programs from proofs.

Outline

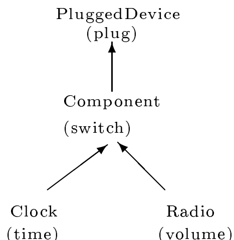
A brief overview of C++ multiple inheritance

A concrete implementation

Results

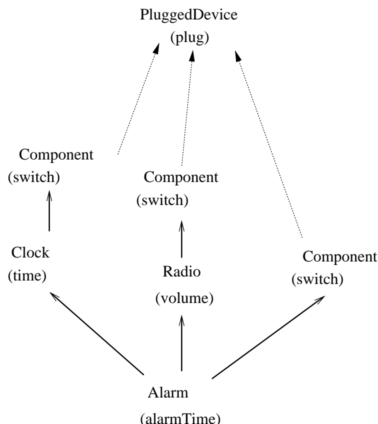
Conclusion and perspectives

Single inheritance



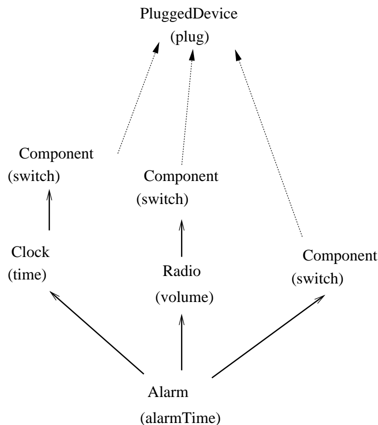

```
struct PluggedDevice {
  int plug;
}

struct Component : PluggedDevice {
  int switch;
}

struct Clock : Component {
  int time;
}

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

Two kinds of multiple inheritance



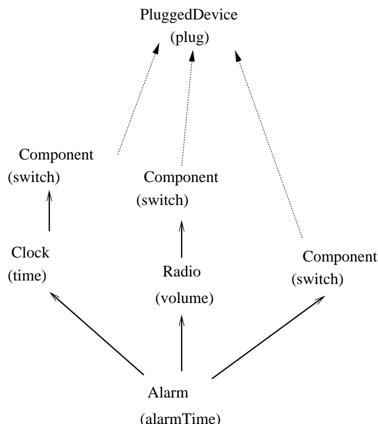

```
struct PluggedDevice {
  int plug;
}

struct Component : virtual PluggedDevice {
  int switch;
}

struct Clock : Component {
  int time;
}

struct Radio : Component {
  int volume;
}

struct Alarm : Clock, Radio, Component {
  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

Formalization : abstract object representation

```
Inductive value : Set :=
  | ...
  |  Ref (heapBlockID * list ident)
  | ...
Record object : Set := makeObject {
  class  : ident;
  fields : list
   (list ident * FieldSignature.t
     * value)
}.
Variable heap : heapBlockID -> option object.
```

Formalization : abstract object representation

```
Inductive value : Set :=
  | ...
  | Ref (heapBlockID * list ident)
  | ...
Record object : Set := makeObject {
  class  : ident;
  fields : list
   (list ident * FieldSignature.t
     * value)
}.
Variable heap : heapBlockID -> option object.
```

Formalization : abstract object representation

```
Inductive value : Set :=
  | ...
  |  Ref (heapBlockID * list ident)
  | ...
Record object : Set := makeObject {
  class  : ident;
  fields : list
   (list ident * FieldSignature.t
     * value)
}.
Variable heap : heapBlockID -> option object.
```

Abstract object representation : field access and cast

```
Alarm * alarm = ... ;

Radio * radio =
 static_cast<Radio *>(alarm);

int i = radio->volume;
```

```
Let alarm :=
    Ref (someBlockID,
         classAlarm :: nil).

Let radio :=
    Ref (someBlockID,
         classAlarm :: classRadio :: nil).

Let i :=
  List.assoc
    (classAlarm :: classRadio :: nil,
     fieldVolume)
  (heap someBlockID).fields.
```

```
PluggedDevice * pda =
  static_cast<PluggedDevice *>(alarm);

PluggedDevice * pdr =
  static_cast<PluggedDevice *>(radio);
```

```
Let pda :=
  Ref (someBlockID,
       classPluggedDevice :: nil).

Let pdr :=
  Ref (someBlockID,
       classPluggedDevice :: nil).
```

Outline

A brief overview of C++ multiple inheritance

A concrete implementation

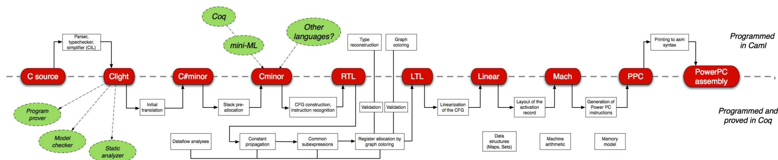
Results

Conclusion and perspectives

Our goal

- Choose a concrete implementation for object layout
- Formalize it in Coq using the Compcert memory model
- Formalize a compilation of elementary object operations (field access, cast, method call) to this concrete implementation
- Show that this compilation is sound wrt. abstract object representation and high-level semantics

What is Compcert ?



Leroy et al., since 2005

- A verified compiler from C to PowerPC
- Compiler proved in Coq and obtained by extraction

Theorem (Semantics preservation)

If P_c is a C program and ***if*** *the compiler produces an assembly code P_{PPC}, then any possible behavior of P_C is also a possible behavior of P_{PPC}.*

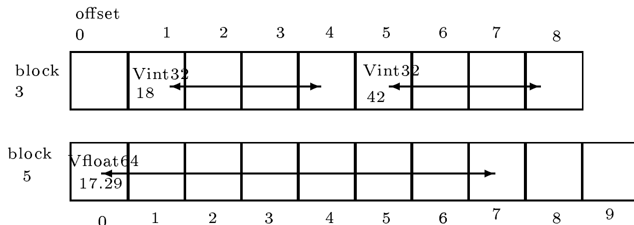
Hypotheses induced by Compcert

- A memory model common to all intermediate languages
- Calling conventions for procedures in intermediate languages
- PowerPC target : 32-bit machine integers and pointer offsets

The Compcert memory model

Leroy and Blazy (2008)

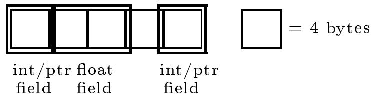
- Memory is a collection of blocks
- Each block is an array of byte cells
- A value can span several byte cells
- A pointer : block ID and offset within this block

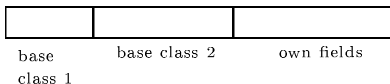


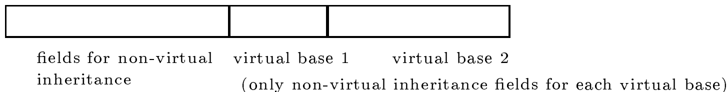
```
Definition load :
  chunk -> mem -> Compcert.blockID -> Z
  -> option Compcert.val := ...
Definition store :
  chunk -> mem -> Compcert.blockID -> Z -> Compcert.val
  -> option mem := ...
Definition chunk_size : chunk -> Z := ...

Theorem load_store_other :
  forall m chunk1 block1 offset1 val m',
    store chunk1 m block1 offset1 = Some m' ->
  forall chunk2 block2 offset2,
    block1 <> block2 \/
    offset1 + chunk_size chunk1 <= offset2 \/
    offset2 + chunk_size chunk2 <= offset1 ->
  load m' chunk2 block2 offset2 =
  load m chunk2 block2 offset2
```

```
Definition load :
  chunk -> mem -> Compcert.blockID -> Z
  -> option Compcert.val := ...
Definition store :
  chunk -> mem -> Compcert.blockID -> Z -> Compcert.val
  -> option mem := ...
Definition chunk_size : chunk -> Z := ...

Theorem load_store_other :
  forall m chunk1 block1 offset1 val m',
    store chunk1 m block1 offset1 = Some m' ->
  forall chunk2 block2 offset2,
    block1 <> block2 \/
    offset1 + chunk_size chunk1 <= offset2 \/
    offset2 + chunk_size chunk2 <= offset1 ->
  load m' chunk2 block2 offset2 =
  load m chunk2 block2 offset2
```

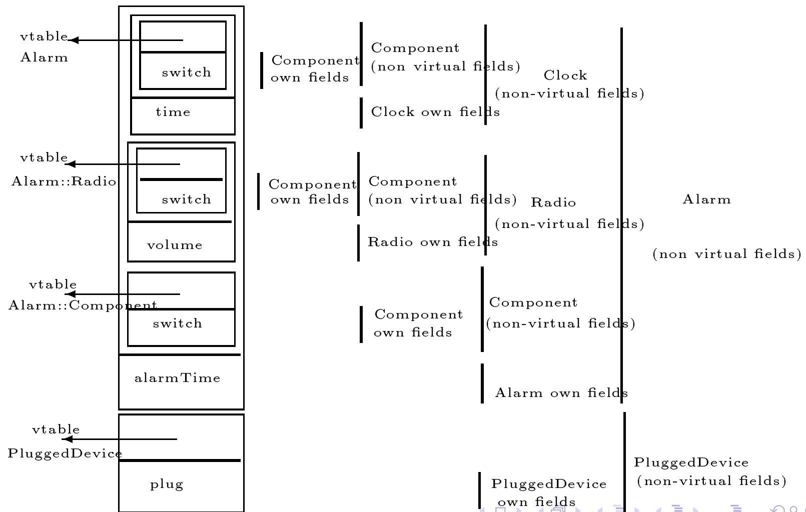
A concrete implementation







A concrete implementation : example



Methodology

- We axiomatize expected properties about object field offsets (which are easily decidable).
- The actual offsets are expected to be computed by an external oracle.
- The output of the oracle may be checked by a formally verified validator.
- The expected properties may leave flexibility wrt. alignment and padding, or to allow the oracle to use a particular strategy.

Own class fields

```
Variable own_offsets : ident -> FieldSignature.t -> Z.

Hypothesis own_offsets_dont_overlap :
  forall class f1 f2,
    f1 <> f2 ->
  own_offsets class f1 + size f1 <= f2 \/
  own_offsets class f2 + size f2 <= f1.

Variable bound : ident -> Z.

Hypothesis own_offsets_le_bound :
  forall class f o,
  own_offsets class f + size f <= bound class.
```

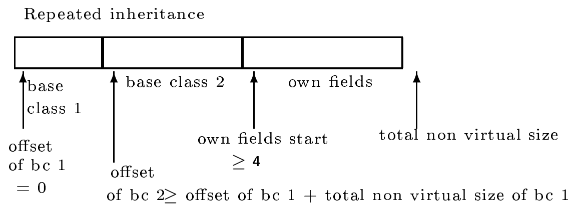
Own class fields

```
Variable own_offsets : ident -> FieldSignature.t -> Z.

Hypothesis own_offsets_dont_overlap :
  forall class f1 f2,
    f1 <> f2 ->
  own_offsets class f1 + size f1 <= f2 \/
  own_offsets class f2 + size f2 <= f1.

Variable bound : ident -> Z.

Hypothesis own_offsets_le_bound :
  forall class f o,
  own_offsets class f + size f <= bound class.
```

Repeated inheritance

- As we already know for each class a bound on the size of own fields (without inheritance), all it remains is to assign an offset for each immediate non-virtual base class of the class.
- This step reserves extra space for pointer to virtual table.
- Optimization : no extra space is reserved for the vtable pointer of the first non-virtual immediate base class ("primary base").

Repeated inheritance



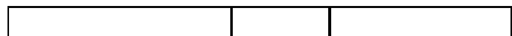
Field offset for non-virtual inheritance

```
Fixpoint non_virtual_offset (path : list ident) : Z :=
match path with
| _::nil  => 0
| a::b::q => offset_of_in b a + non_virtual_offset (b::q)
end.
```

Virtual inheritance

- If D inherits from B, then every virtual base of B is a virtual base of D.
- So it is wise to treat virtual inheritance only when the non-virtual inheritance tree is entirely treated.
- For each class D, the oracle is expected to give an offset (relatively to D) of every virtual base of D.

Virtual inheritance



Virtual offset and field offset

```
Definition virtual_offset
 (cl : ident) (p : list ident) : Z :=
virtual_offset_of_in (first p) cl
+ non_virtual_offset p.

Definition field_offset (cl : ident)
 (p : list ident)
 (f : FieldSignature.t) : Z :=
virtual_offset cl p
+ own_fields_start (last p)
+ own_field_offset (last p) f.
```

A class is never a virtual base of itself, but its offset relatively to itself is considered to be 0, so as to treat non-virtual inheritance.

Virtual offset and field offset

```
Definition virtual_offset
 (cl : ident) (p : list ident) : Z :=
virtual_offset_of_in (first p) cl
+ non_virtual_offset p.

Definition field_offset (cl : ident)
 (p : list ident)
 (f : FieldSignature.t) : Z :=
virtual_offset cl p
+ own_fields_start (last p)
+ own_field_offset (last p) f.
```

A class is never a virtual base of itself, but its offset relatively to itself is considered to be 0, so as to treat non-virtual inheritance.

Compilation of field access

A concrete state holds a concrete memory along with a correspondence between an abstract heap block (abstract object slot) and an offset within the concrete heap in the concrete memory.

```
Record state : Set := make_state {
  heap_block      : Compcert.blockID;
  m               : mem;
  block_matching : heapBlockID -> Compcert.blockID
}.
```

Compilation of field access

```
Definition access_field
    (objRef : heapBlockID)
    (path   : list ident)
    (f : FieldSignature.t)
    (abstract_heap : heap -> option object)
    (s : state)
    option Compcert.val :=
  match abstract_heap objRef with
  | Some object =>
    load (sizeof f) s.m s.heap_block
      (s.block_matching objRef
       + field_offset object.class path f)
  | None => None
  end.
```

Outline

A brief overview of C++ multiple inheritance

A concrete implementation

Results

Conclusion and perspectives

Matching values

```
Inductive match_values
  (abstract_heap : heapBlockID -> option object)
  (s : state) :
  value -> Compcert.val -> Prop :=
| ...
| match_value_ref : forall objRef object path,
    abstract_heap objRef = Some object ->
    offset = MachineInteger.repr (
      s.block_matching objRef
      + virtual_offset object.class path
    ) ->
    match_values
      (Ref   objRef path)
      (Vptr s.heap_block offset)
```

Soundness of field access

Theorem

The following invariant :

```
forall abstract_heap objRef object,
  abstract_heap objRef = Some object ->
forall path field absval,
  List.assoc (path, field) object.fields = Some absval ->
forall state, exists conval,
  access_field objRef path field abstract_heap state
   = Some conval
  /\  match_values abstract_heap state absval conval.
```

holds when a field is modified.

Good fields property

The most technical lemma.

```
Theorem fields_do_not_overlap :
  forall cl : ident,
  forall p1 p2 : list ident,
  forall f1 f2 : FieldSignature.t,
    (p1, f1) <> (p2, f2) ->
  forall o1 o2,
    field_offset cl p1 f1 + size f1
    <= field_offset cl p2 f2
    \/
    field_offset cl p2 f2 + size f2
    <= field_offset cl p1 f1.
```

Proof sketch

- `fields_do_not_overlap` proved step by step : first consider two fields of the same class (trivial), then two fields in the non-virtual inheritance tree, then two fields in the whole inheritance tree.
- Additional bounding properties about field offsets are necessary to convert $\mathbb{Z}$ integers into machine integers.

Vtable pointers

The same way, we show :

Theorem

Field modification does not change pointers to virtual tables.

Casts

- Static casts without virtual inheritance OK (arithmetics).
- Upcasts with virtual inheritance need additional hypotheses on virtual tables.

Dynamic method dispatch

- Dynamic dispatch needs hypotheses on virtual tables.
- Thunks are currently not supported : Compcert and its intermediate languages seem to model no convenient way of optimizing them (e.g. functions with multiple entry points)
- **this** pointer adjustment offset assumed present along with method pointer in the virtual table. Under such hypotheses, OK. However, it is costly (additional memory access needed at each method call, even though constant-time).

Outline

A brief overview of C++ multiple inheritance

A concrete implementation

Results

Conclusion and perspectives

What has been done

- Our choice of concrete object layout is realistic insofar as it takes constant-time field access and static casts into account.
- Our choice of object layout is sound wrt field access (read/write).
- All static casts are sound without virtual inheritance.

What still remains to do

- Optimization of virtual method call : maybe use of optimized "tail call" ?
- Object construction and destruction (virtual tables are not the same during construction as during the "normal" life of the object)
- Dynamic cast : either use a switch, or formalize RTTI

Our work, a first step towards...

- A verified compiler based on Compcert (Work in progress from a subset of C++ to the RTL intermediate language, an assembly-like language with an unbounded number of registers).
- Formal verification of real-world Application Binary Interfaces (ABI)

Thank you !

In Texas there are lots of fields...