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

Tahina Ramananandro[1] Gabriel Dos Reis[2] Xavier Leroy[1]

[1]INRIA Paris-Rocquencourt [2]Texas A&M University

January 27th, 2012

Construction and destruction

- In most object-oriented languages: constructors attached to classes for object initialization
  - Turn “raw memory” into an object
  - Establish invariants required to manipulate the object, e.g. acquire necessary resources

Construction and destruction

- In most object-oriented languages: constructors attached to classes for object initialization
  - Turn "raw memory" into an object
  - Establish invariants required to manipulate the object, e.g. acquire necessary resources

- Symmetrically to constructors, C++ introduces destructors:
  - Turn the object back to "raw memory"
  - **Resource management**: release acquired resources

$\rightarrow$ **Resource acquisition is initialization** (RAII)

Construction and destruction in C++

```
struct File {
  FILE* handle;
  void write(char* string) { ... }

  // Constructor
  File(char* name): handle(fopen(name, "w")) {}

  // Destructor
  ~File()                { fclose(handle); }
};

int main(int argc, char* argv[]) {
  File f("toto.txt");
  f.write("Hello world!");
  // automatic destructor call on scope exit
  // Resource acquisition is initialization (RAII)
  return 0;
}
```

Construction and destruction in C++

```
struct File {
   FILE* handle;
   void write(char* string) { ...  }

   // Constructor
   File(char* name):  handle(fopen(name, "w")) {}

   // Destructor
   ~File()                  { fclose(handle); }
};

int main(int argc, char* argv[]) {
   File f("toto.txt");
   f.write("Hello world!");
   // automatic destructor call on scope exit
   // Resource acquisition is initialization (RAII)
   return 0;
}
```

Construction and destruction in C++

```
struct File {
   FILE* handle;
   void write(char* string) { ...  }

   // Constructor
   File(char* name):  handle(fopen(name, "w")) {}

   // Destructor
   ~File()                  { fclose(handle); }
};

int main(int argc, char* argv[]) {
   File f("toto.txt");
   f.write("Hello world!");
   // automatic destructor call on scope exit
   // Resource acquisition is initialization (RAII)
   return 0;
}
```

Resource management and embedded objects


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

C++ guarantees that two subobjects of the same object be destructed in the reverse order of their construction.

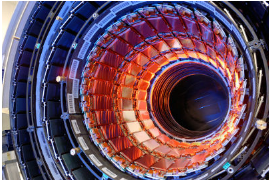
C++ construction/destruction principles

- Subobjects must be constructed and destructed only once
- Subobjects of an object must be destructed in the reverse order of their construction
- Operations must not rely on object parts that are not yet constructed (or already destructed)

C++ in critical software

Those aspects commonly arise in critical C++ software:

Joint Strike Fighter



Large Hadron Collider

Mars Rover

Our contribution

- A formal operational semantics for C++ object construction and destruction
- Captures and clarifies a number of delicate points, such as interaction with multiple (including virtual) inheritance
- Allows to state and prove high-level properties expected by the programmers, such as RAII
- Mechanized in Coq

Outline

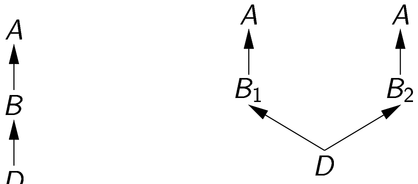
1. Construction and destruction within the C++ object model
2. Formal semantics
3. Reasoning about the semantics
4. Assessment

C++ multiple inheritance

$$\begin{tikzcd} A \\ B \arrow[u] \\ D \arrow[u] \end{tikzcd}$$

Single inheritance

C++ multiple inheritance

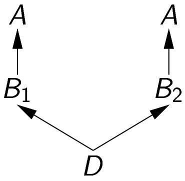


Single inheritance

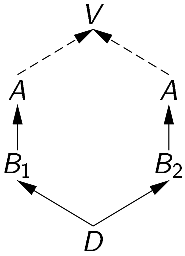
Non-virtual multiple inheritance

C++ multiple inheritance

Single inheritance



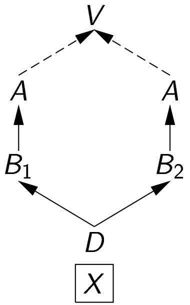
Non-virtual multiple inheritance



Virtual multiple inheritance

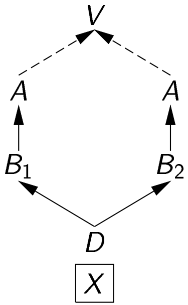
Subobjects must be constructed only once

Subobject construction order

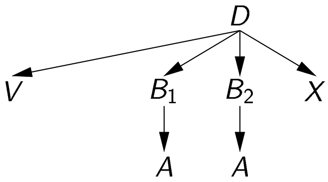


Class hierarchy

Subobject construction order

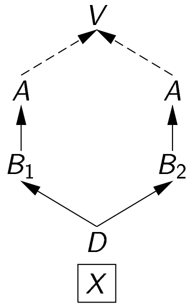


Class hierarchy

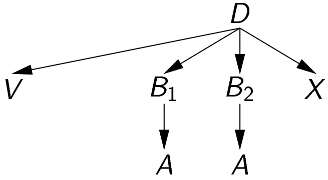


Construction tree

Subobject construction order



Class hierarchy



Construction tree

- Construction in depth-first postfix left-to-right traversal of construction tree
- Destruction in reverse order of construction: depth-first prefix right-to-left

Outline

The C++ object construction protocol

```
type kind = MostDerived | Base

let rec construct kind obj =

  if kind = MostDerived then
    for each direct or indirect virtual base V in inheritance graph order:
      construct Base (subobj_V obj)
    end for
  end if;
  for each direct non-virtual base B:
    construct Base (subobj_B obj)
  end for;

  for each field f:
    construct MostDerived (field_f obj)
  end for;
  execute constructor body
```

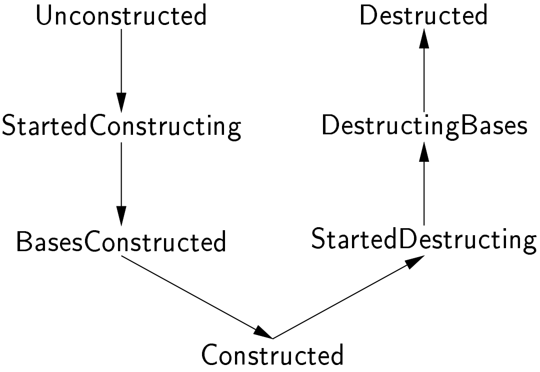
The C++ object construction protocol

```
type kind = MostDerived | Base

let rec construct kind obj =
  obj.constrState ← StartedConstructing;
  if kind = MostDerived then
    for each direct or indirect virtual base V in inheritance graph order:
      construct Base (subobj_V obj)
    end for
  end if;
  for each direct non-virtual base B:
    construct Base (subobj_B obj)
  end for;
  obj.constrState ← BasesConstructed;
  for each field f:
    construct MostDerived (field_f obj)
  end for;
  execute constructor body ;
  obj.constrState ← Constructed
```

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.

Single-stepping the semantics

- Reasoning about resource management needs comparing the construction states of:
  - two objects at one execution point
  - one object at two execution points
- Reference interpreter / big-step semantics not convenient
  - Much information hidden in call stack
- We use a small-step semantics with explicit continuations.

General shape of the semantics

Transitions

$$(P, K, \mathcal{G}) \to (P', K', \mathcal{G}')$$

General shape of the semantics

Transitions

$$(P, K, \mathcal{G}) \to (P', K', \mathcal{G}')$$

State

$$\mathcal{G}, \mathcal{G}' ::= (subobject.field \mapsto value) \times (subobject \mapsto constrstate)$$

General shape of the semantics

Transitions

$$(P, K, \mathcal{G}) \to (P', K', \mathcal{G}')$$

State

$$\mathcal{G}, \mathcal{G}' ::= (\mathit{subobject}.\mathit{field} \mapsto \mathit{value}) \times (\mathit{subobject} \mapsto \mathit{constrstate})$$

Control point

$$\begin{array}{rcll}
\beta & ::= & \text{Bases(Virtual)} \mid \text{Bases(DirectNonVirtual)} \mid \text{Fields} & \\
P, P' & ::= & \text{Codepoint}(\mathit{stmt}, \dots) & \text{executing a statement} \\
& \mid & \text{ConstrMostDerived}(\mathit{this}, D, \dots) & \text{Constructing most-derived object}\mathit{this} \\
& \mid & \text{Constr}(\mathit{this}, \beta, L, \dots) & \text{Constructing subobjects of } \mathit{this} \\
& \mid & \text{DestrMostDerived}(\mathit{this}, D) & \text{Destructing most-derived object}\mathit{this} \\
& \mid & \text{Destr}(\mathit{this}, \beta, L) & \text{Destructing subobjects of } \mathit{this}
\end{array}$$

General shape of the semantics

Transitions

$$(P, K, \mathcal{G}) \to (P', K', \mathcal{G}')$$

State

$$\mathcal{G}, \mathcal{G}' ::= (\mathit{subobject.field} \mapsto \mathit{value}) \times (\mathit{subobject} \mapsto \mathit{constrstate})$$

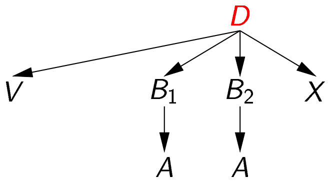
Control point

β	$::=$	Bases(Virtual) | Bases(DirectNonVirtual) | Fields	
P, P'	$::=$	Codepoint($\mathit{stmt}, \dots$)	executing a statement
	|	ConstrMostDerived($\mathit{this}, D, \dots$)	Constructing most-derived object*this*
	|	Constr($\mathit{this}, \beta, L, \dots$)	Constructing subobjects of *this*
	|	DestrMostDerived(this, D)	Destructing most-derived object*this*
	|	Destr(this, β, L)	Destructing subobjects of *this*

Continuations: "what to do next"

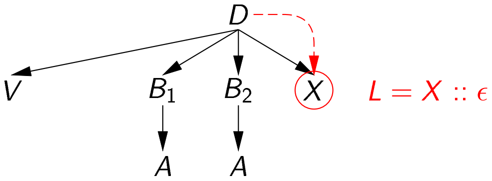
K, K'	$::=$	Kreturn($\mathit{stmt}, \dots, K'$)	return from function
	|	KconstrMostDerived($\mathit{this}, D, \dots, K'$)	Construct most-derived object once initializer is done
	|	Kconstr($\mathit{this}, \beta, B, L, \dots, K'$)	Construct subobject B once initializer is do
	|	Kconstrother($\mathit{this}, \beta, B, L, \dots, K'$)	Construct remaining subobjects
	|	Kdestr(this, K')	Destruct all subobjects of *this* once destructor body is done
	|	Kdestrother($\mathit{this}, \beta, B, L, K'$)	Destruct remaining subobjects of *this*
	|	KdestrMostDerived(this, D, K')	Remember to destruct virtual bases of most-derived object

Some destruction rules



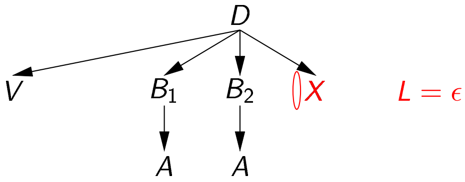
$$\frac{\sim D()\{stmt\} \qquad Env = \varnothing[\mathsf{this} \leftarrow \pi] \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{StartedDestructing}]}{\begin{array}{l}(\mathsf{DestrMostDerived}(\pi, D), \mathcal{K}, \mathcal{G}) \\ \rightarrow (\mathsf{Codepoint}(stmt, \epsilon, Env, \epsilon), \\ \quad \mathsf{Kdestr}(\pi) :: \mathsf{KdestrMostDerived}(\pi, D) :: \mathcal{K}, \mathcal{G}')\end{array}}$$

Some destruction rules



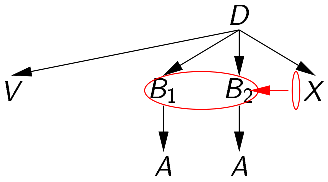
$$\frac{\pi : D \qquad L = \mathrm{rev}(\mathcal{F}(D))}{\begin{array}{l} \quad (\mathrm{Codepoint}(\mathrm{return}, \mathit{Stmt}^*, \mathit{Env}, \epsilon), \mathrm{Kdestr}(\pi, K), \mathcal{G}) \\ \rightarrow (\mathrm{Destr}(\pi, \mathrm{Fields}, L), K, \mathcal{G}) \end{array}}$$

Some destruction rules



$$
\frac{\pi' = \pi.X}{\begin{array}{l} \quad (\mathrm{Destr}(\pi, \mathrm{Fields}, X :: L), \mathcal{K}, \mathcal{G}) \\ \to (\mathrm{DestrMostDerived}(\pi'), \\ \qquad \mathrm{Kdestrother}(\pi, \mathrm{Fields}, X, L) :: \mathcal{K}, \mathcal{G}) \end{array}}
$$

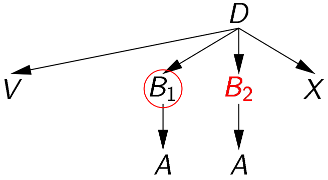
Some destruction rules



$L = B_2 :: B_1 :: \epsilon$

$$\frac{L = \mathrm{rev}(\mathcal{DNV}(D)) \qquad \begin{array}{c}\pi : D \\ \mathcal{G}' = \mathcal{G}[\mathrm{ConstrState}(\pi) \leftarrow \mathrm{DestructingBases}]\end{array}}{\begin{array}{l}(\mathrm{Destr}(\pi, \mathrm{Fields}, \epsilon), \mathcal{K}, \mathcal{G}) \\ \to (\mathrm{Destr}(\pi, \mathrm{Bases}(\mathrm{DirectNonVirtual}), L), \mathcal{K}, \mathcal{G}')\end{array}}$$

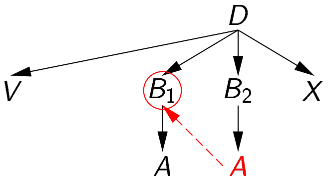
Some destruction rules



$L = B_1 :: \epsilon; \beta = \text{DirectNonVirtual}$

$$\frac{\sim B_2()\{\mathit{stmt}\} \qquad \pi' = \text{AddBase}(\pi, \beta, B_2) \\ \mathit{Env} = \varnothing[\text{this} \leftarrow \pi'] \qquad \mathcal{G}' = \mathcal{G}[\text{ConstrState}(\pi') \leftarrow \text{StartedDestructing}]}{\begin{array}{l} (\text{Destr}(\pi, \text{Bases}(\beta), B_2 :: L), \mathcal{K}, \mathcal{G}) \\ \to (\text{Codepoint}(\mathit{stmt}, \epsilon, \mathit{Env}, \epsilon), \\ \quad \text{Kdestr}(\pi') :: \text{Kdestrother}(\pi, \text{Bases}(\beta), B_2, L) :: \mathcal{K}, \mathcal{G}') \end{array}}$$

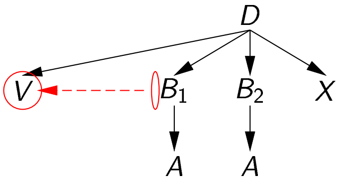
Some destruction rules



$L = B_1 :: \epsilon; \beta = \text{DirectNonVirtual}$

$$\frac{\mathcal{G}' = \mathcal{G}[\text{ConstrState}(\pi) \leftarrow \text{Destructed}]}{\begin{array}{l}(\text{Destr}(\pi, \text{Bases}(\text{DirectNonVirtual}), \epsilon), \\ \quad \text{Kdestrother}(\pi', \text{Bases}(\beta), B_2, L) :: \mathcal{K}, \mathcal{G}) \\ \to (\text{Destr}(\pi', \text{Bases}(\beta), L), \mathcal{K}, \mathcal{G}')\end{array}}$$

Some destruction rules



$L = V :: \epsilon$

$$\frac{L = \mathrm{rev}(\mathcal{VO}(D))}{\begin{array}{l} \quad (\mathrm{Destr}(\pi, \mathrm{Bases}(\mathrm{DirectNonVirtual}), \epsilon), \\ \quad\ \ \mathrm{KdestrMostDerived}(\pi, D) :: \mathcal{K}, \mathcal{G}) \\ \to (\mathrm{Destr}(\pi, \mathrm{Bases}(\mathrm{Virtual}), L), \mathcal{K}, \mathcal{G}) \end{array}}$$

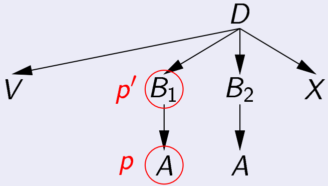
Outline

Low-level properties

Lemma (Parent and child construction states)

If p is a child of p' in the construction tree, 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

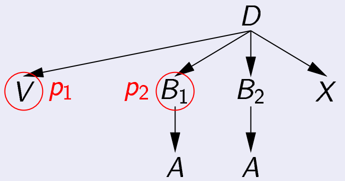


Low-level properties

Lemma (Sibling construction states)

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



High-level properties

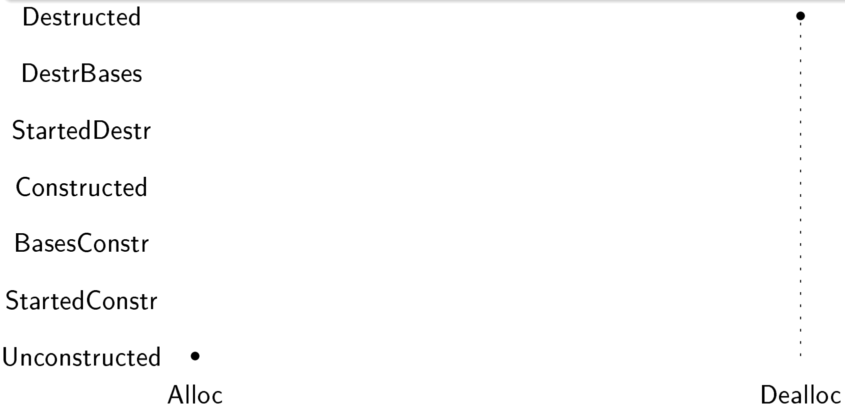
Theorem (Resource Acquisition is Initialization)

Consider a block `{ C x; ... }` *. When it exits, x and all its subobjects were constructed exactly once, then destructed exactly once.*

High-level properties

Theorem (Resource Acquisition is Initialization)

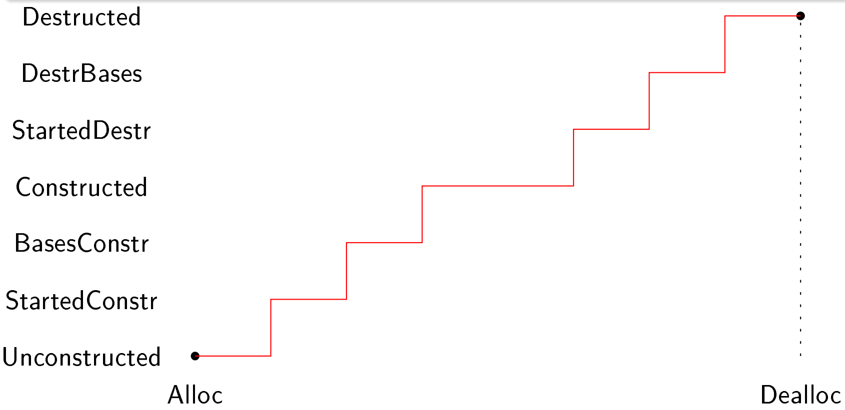
Consider a block `{ C x; ... }`. *When it exits, x and all its subobjects were constructed exactly once, then destructed exactly once.*



High-level properties

Theorem (Resource Acquisition is Initialization)

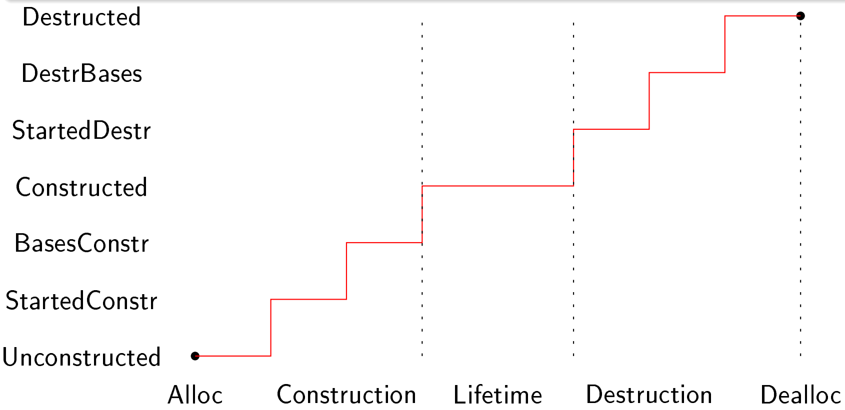
Consider a block `{ C x; ... }` *. When it exits,* x *and all its subobjects were constructed exactly once, then destructed exactly once.*



High-level properties

Theorem (Resource Acquisition is Initialization)

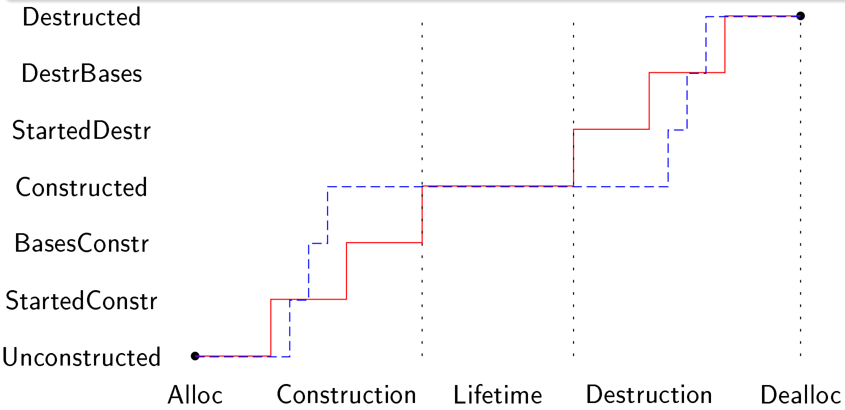
Consider a block `{ C x; ... }` *. When it exits, x and all its subobjects were constructed exactly once, then destructed exactly once.*



High-level properties

Theorem (Resource Acquisition is Initialization)

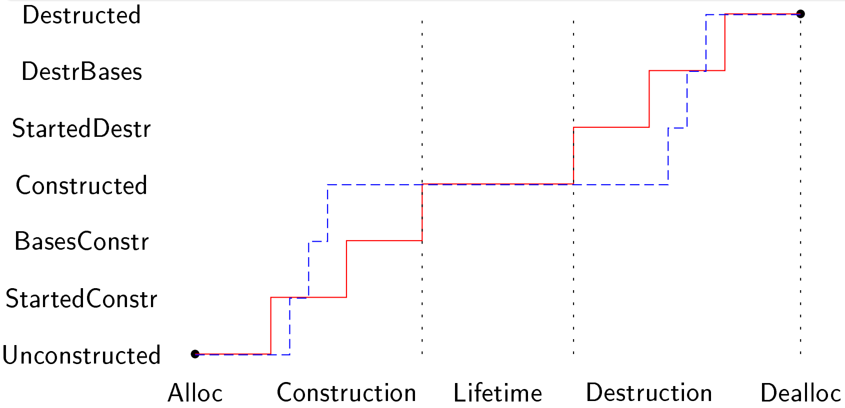
Consider a block `{ C x; ... }` *. When it exits, x and all its subobjects were constructed exactly once, then destructed exactly once.*



High-level properties

Theorem (Resource Acquisition is Initialization)

Consider a block `{ C x; ... }` *. When it exits, x and all its subobjects were constructed exactly once, then destructed exactly once.*



More high-level properties

Proved in Coq

Theorem

The lifetime of an object is included in the lifetimes of all its subobjects. In other words, if an object is constructed, then all its subobjects are constructed.

Theorem

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

Theorem (Progress)

If constructors (resp. destructors) are trivial, then the construction (resp. destruction) process always succeeds (the semantics is well-defined).

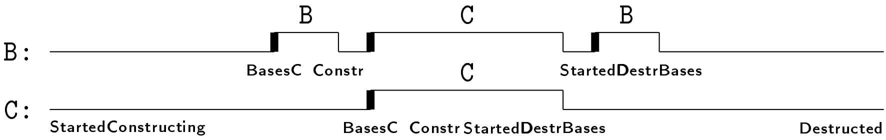
Virtual functions during construction and destruction

```
struct B {
  virtual void f () {...}
  B () {
    this->f (); // always calls B::f()
  }
};

struct C : B {
  virtual void f () {...}
  C () : B () {
    this->f (); // always calls C::f()
  }
};
```

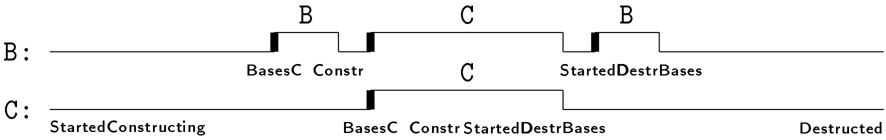
The generalized dynamic type of a subobject

```
struct C : B { ... };
```



The generalized dynamic type of a subobject

```
struct C : B { ... };
```



Definition

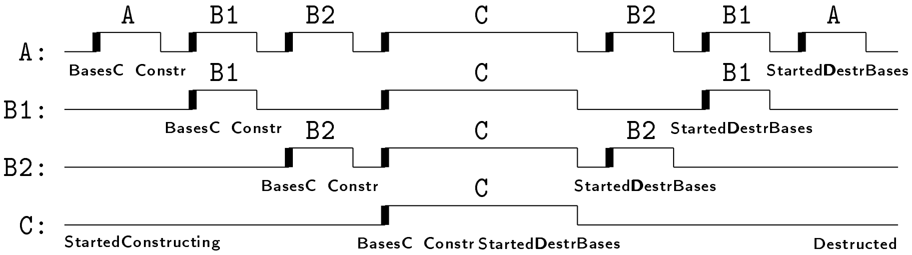
A subobject σ has a generalized dynamic type $\sigma_\circ$ if, and only if:

- *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 $\sigma_\circ$*

Considered as the most-derived object for polymorphic operations (dynamic cast, virtual function call)

The generalized dynamic type of a subobject

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



Application: A verified compiler

Our language (Constructors Destructors) $\rightarrow$ CoreC++ (Wasserrab & al.) + Set dynamic type $\rightarrow$ CompCert Cminor (Leroy & al.) (low-level memory) + virtual tables & VTT

- Popular compilation scheme + one optimization: separated constructors for most-derived object and inheritance subobject
- Proof of semantic preservation (forward simulation)

$$\begin{tikzcd} s_1 \arrow[r] \arrow[d, dash, "\text{invariant}"] & s_2 \arrow[d, dash, dotted, "\text{invariant}"] \\ s_1' \arrow[r, dotted, "+"] & s_2' \end{tikzcd}$$

(cf. Ramananandro's Ph. D. thesis, Chapter 11)

Outline

1. Construction and destruction within the C++ object model
2. Formal semantics
3. Reasoning about the semantics
4. Assessment

Conclusions

- A general formal model for C++ construction and destruction
- First to capture correctly interaction with virtual function dispatch
- First machine-checked formalization of RAII

Practical impact

Positive feedback from C++ Standard Committee: uncovered 4 inconsistencies and omissions in the C++03 standard:

- virtual functions during destruction: fixed in C++11
- object lifetime and trivial destructors (pending)
- lifetime of arrays (pending)
- unification of destruction model for built-in types (pending)

Perspectives

Extending the semantics:

- Free store
- C++ copy semantics and temporary objects (passing constructor arguments by value, copy constructor, functions returning structures)
- Exceptions
- Templates

Thank you for your attention

- Coq development fully available on the Web:
  `http://gallium.inria.fr/~tramanan/cxx/construction`
- For further information: `ramanana@nsup.org`

Virtual functions during destruction

```
struct A {
  virtual void f() { ... }
};

struct X {
  A* a;
  X(A* a0): a(a0)  { a->f(); /* defined during
                                construction of C */ }
  ~X()             { a->f(); /* UNDEFINED during
                                destruction of C  */ }
};

struct C: A {
  X x;
  C(): x(this)     {}
};
```

The generalized dynamic type is attached to the object

```
struct A {
  virtual void f() { ... }
};
struct X {
  A* a;
  void g()          { a->f(); }
  X(A* a0): a(a0)   { g();       /* B::f */ }
};
struct B: A {
  X x;
  B(): x(this)      {}
};
struct D: B {
  D(): B()          {}
  virtual void f() {...}
};
int main(int argc, char* argv[]) {
  D d;
  d.x.g();                      /* D::f */
}
```

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.

A core language

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

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

A core language

$$
\begin{array}{rcll}
\mathit{Funct} & ::= & \texttt{virtual}\ f(\mathit{var},\ldots)\{\mathit{Stmt}\} & \text{Virtual function} \\
\mathit{Finit}_m & ::= & & \text{Data member} \\
 & & & \text{initializers} \\
 & & m\{\mathit{Init}_A \ldots\} & \text{Structure} \\
 & \mid & m(\mathit{Stmt}, \mathit{var}) & \text{Scalar} \\
 & & & \\
\mathit{Constr}_C & ::= & C(\mathit{var},\ldots) : \mathit{Init}_{B1},\ldots,\mathit{Init}_{V1},\ldots, & \text{Constructor} \\
\mathit{Destr}_C & ::= & \tilde{}C()\{\mathit{Stmt}\} & \text{Destructor} \\
 & & \mathit{Finit}_m,\ldots\{\mathit{Stmt}\} & \\
 & & & \\
\mathit{Class} & ::= & \texttt{struct}\ C : B1,\ldots,\texttt{virtual}\ V1,\ldots & \text{Class definition} \\
 & & \{\mathit{Constr}_C \ldots; \mathit{Funct} \ldots; \mathit{Destr}_C\} & \\
\mathit{Prog} & ::= & \mathit{Class} \ldots & \text{Program}
\end{array}
$$

The Coq development

47 semantic rules (incl. 15 constr., 12 destr.)

Theories	**Specs loc**	**Proofs loc**
Class hierarchies	996	1308
Well-formed hierarchies	283	1794
Core language	895	144
Invariant	693	81
Invariant preservation	324	13154
High-level properties	2296	6306
Total	**5487**	**22787**

Invariant preservation requires $2\frac{1}{4}$ hours for Coq to check.

Ctxt File Prin Crit Summ Inhe Ordr Prot Csta Step Shap Rulz Chld Sibl Raii High Vfdc Gdyn Cpil Cncl Stan Futu Thnx