Unsound 2022

The four horsemen of OO verification
Is OO verification completely doomed?

Marco Servetto, VUW ECS

History

- Hoare logic / Hoare triple
- {P} C {Q} //P = premise, C = command, Q = postcondition
- Commands are expressed in a minimal ‘while’ language, with if / while / variable-update. (Turing complete)
- Those variables sit in a ‘global memory’ and they are integers or boolean.
- Primitive types only: Crucially the semantic of integers and booleans is fixed and is not dependent of the commands.

History

- Hoare logic extensions
- More primitive data types, for example arrays of integers
- Recursion: multiple functions and function calls.
  - Partial correctness: For every function, we prove that its {P}C{Q} holds assuming all the other functions are correct
  - Total correctness: prove that all recursions and iterations are terminating

- Not OO yet:
  - No subtyping
  - No user defined types
- Many attempts to support full OO.
  - but it is still unobvious how to do it (see the rest of the talk)

The current narrative

- In the current narrative, the two big issues are:
  - aliasing / mutation (often with separation logic to support reasoning)
  - Violations of the LSP (Liskov substitution principle)
- So most researchers in OO verification focus their time and attentions on those two points.
- While those two points are important, I argue that there more fundamental open issues.
- NOTE: all of the 4 points I'm making are well known (by some) but the presentation is novel and compelling.
  Many novices are unaware of those difficulties!

Functional verification

- I will now discuss a few challenges for sound OO verification.
- **Q:** But COQ seems sound, why is it so different in that context?
  - Functional type-based verification languages
    (COQ / AGDA / ISABELLE / LEAN)
    solve those challenges with positivity checking.
- **Q:** So, we can just do what they do but in OO!
  - I will show later then positivity checking restrictions are unsuitable to OO programming.

OO termination analysis can not be modular

OO termination analysis can not be modular

- Objective 1: the surface verification language allows to annotate methods with Total/NonTotal
- Objective 2: independent pieces of code should be independently verifiable, and putting together verified code produces verified code.

- I will now show you that those two objectives are incompatible

3 companies working independently

```
//company0 does I
interface I{
  /*Total*/int a(I i);
  /*Total*/int b(I i);
}
//I verifies: there is nothing to verify

//company1 uses I and does C1
class C1 implements I{
  /*Total*/public int a(I i){ return 5; }
  /*Total*/public int b(I i){ return i.a(new C1()); }
}
//should C1 verify?
// method C1.a(i) clearly terminates.
// method C1.b(i) simply calls a terminating method
```

3 companies working independently

```
interface I{  /*Total*/int a(I i);  /*Total*/int b(I i);  }
class C1 implements I{
  /*Total*/public int a(I i){ return 5; }
  /*Total*/public int b(I i){ return i.a(new C1()); }
}
//company2 uses I and does C2
class C2 implements I{
  /*Total*/public int a(I i){ return i.b(new C2()); }
  /*Total*/public int b(I i){ return 5; }
}
//Should C2 verify? it is symmetric to C1… however
      new C1().b(new C2());  //this goes in loop!
```

When it is safe to do a call?

- How does `new C1().b(new C2());` loops?
- There are no while/fors.
- Is there recursion?
- Where?
- “I” has no recursion
- “C1” does not look recursive, just method b calls method a.
- “C2” is the same…
- However...

```
interface I{  int a(I i);  int b(I i);  }
class C1 implements I{
  int a(I i){ return 5; }
  int b(I i){ return i.a(new C1()); }
}
class C2 implements I{
  int a(I i){ return i.b(new C2()); }
  public int b(I i){ return 5; }
}
```

Solution?

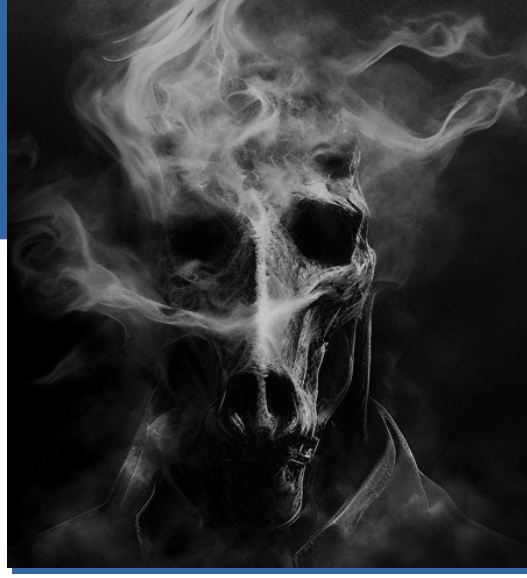
- I have no idea, probably we would need a richer annotation. My example shows that just ‘Total/NotTotal’ is not sufficient.

- We may need to expose ‘termination layers’ or something like that to the user.

- If an OO verification language supports just ‘Total/NotTotal’ then it is **NOT** doing modular verification. Adding apparently unrelated code could break termination of already verified code.

Ghost + partial correctness = unsound

Ghost + partial correctness = unsound



```
interface Nope{
  /*ensures false*/int whoops();
}

class Omega{
  Nope omega(Omega o){ return o.omega(o); }
}

...

Omega o=new Omega();
/*ghost*/ int nope = o.omega(o).whoops();
//now 'false' holds
assert 1==2;//verifies!
```

Ghost + partial correctness = unsound

Takes a little more effort to convince Dafny, but the concept is the same.

```
trait Nope { function method whoops(): () ensures false }

class Omega {
  const omega: Omega -> Nope
  constructor(){ omega := (o: Omega) => o.omega(o); }
}

method test() ensures false {
 var o := new Omega();
 ghost var _ := o.omega(o).whoops();
 //false holds here!
 assert 1==2;
 }
```

.

Ghosts

- If a verifier has ghost state and offers only partial correctness, or it has a bug in the termination checker, then it is unsound.

- Conventionally ghost statements are NOT considered part of the actual program and are not executed at run time.
- If the ghost statement was to actually run as part of the actual program (as in Ada/Spark) then it would not reach the ‘false’ state, it would instead go in loop at run time.

Well founded contracts?

Well founded contracts?

- The simplest OO program? The empty program!
  ok, after that?
  An interface with a single abstract method!
  ok, after that?
  Well, the method could take an argument!

```
interface C{ bool comm(C c); }
```

- Ok, so.. what could be the simplest contract I could write for this method? Return a constant!
  ok, after that? The method is commutative!

```
interface C{/*ensures this.comm(c)==c.comm(this)*/bool comm(C c); }
```

- Can we try to verify a simple implementation?

Well founded contracts?

```
interface C{/*ensures this.comm(c)==c.comm(this)*/bool comm(C c); }

class CTrue implements C{

  bool comm(C c){ return true; }

  //proof of this.comm(c)==c.comm(this)
  //how to do it? we would need to know what c.comm(this) would do.
  //Standard solution: we use the contract of C.comm!

  //step1: we know that c.comm(this) respects the postcondition
  //step2: we rewrite the equality so that c.comm(this)
  //       gets rewritten into this.comm(c)
  //step3: reflexivity over this.comm(c)==this.comm(c)  QED

  //Unconvincing: we did not even needed to look inside the body!

}
```

Well founded contracts?

```
interface C{/*ensures this.comm(c)==c.comm(this)*/bool comm(C c); }
class CTrue  implements C{ bool comm(C c){ return true; } }
class CFalse implements C{ bool comm(C c){ return false; } }
…
new CTrue().comm(new CFalse()) != new CFalse().comm(new CTrue());
```

- Broken again!
- But here, the code does not go in loop!
- Is the contract going in ‘loop’?
  It depends of what you mean by ‘loop’
- In static verification the contracts can even be specified in a logical metalanguage that could not be executed, not even in principle.
  So, what does it even means to go in ‘loop’ for a contract?

Well founded contracts

- Those kinds of contracts are often called ‘ill founded’
- However, it is not fully clear what does it takes for a contract to be well/ill-founded.
- I suspect that modular well foundness checking is impossible.
- Similar to the impossible modular termination checking.

- We may have to further annotate our contracts in some way. Or, it may be a fundamentally impossible problem.

Well founded contracts

- It may be a fundamentally impossible problem.
- Consider the following variant:

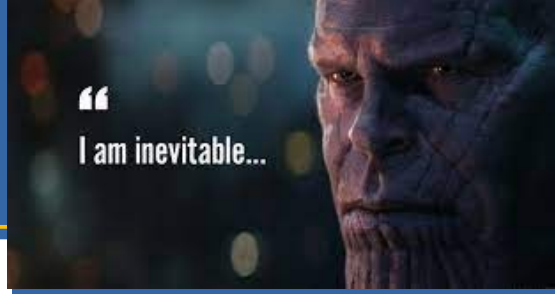
```
interface CA{
  /*ensures this.comm(c)==c.m(this)*/bool comm(CB cb); }

interface CB{
  /*ensures ..something..*/ bool m(CA a); }
```

- Is this a well founded contract?
  Does it just depends on what '..something..' is?
- What if a class implementing CB was to use CA inside?
- What uses would be ok? What would not? Why?

Inevitable reduction to primitive operations

Inevitable reduction to primitive operations



- Is most OO verification just ‘an extra layer’ over verification on primitive types? If not, what can we verify in a basic OO system that does not have int/bools? For example, Featherweight Java.
- How would a contract even look if we do not have == or any int/bool operation?
- What can we ‘observe’?
- In pure OO, objects have methods, methods return other objects… that have other methods.
- How do we express that the method did the right thing?

Inevitable reduction to primitive operations

- Idea: allows for == just in the specification. For example

```
class B{}

class A{
  B b; A(B b){this.b=b;}
  /*ensures this.getB()==new B()*/
  B getB(){return b;}
}
```

- Bad Idea! OO design encourages returning objects that behave like a certain object but they are a wrapper doing some twists. This fundamental pattern could not be captured in this way.

Inevitable reduction to primitive operations

Objective: we should have a modular and sound verification system for some functional variation of FJ before we can hope to get a sound verification system that works with aliasing and mutation.

- In such a language we should be able to express and verify
  - Pure OO encoding of Booleans and Peano numbers
  - Generic Linked lists and stream-like operations on those lists
  - Most of the OO patterns that can be expressed in a functional setting: Decorators, visitors, proxy, chain of responsibility, etc.

Until we have that, I think it is unwise to focus on the imperative aspects.

Potential solutions

Positive type dependencies are everywhere in OO programming

The root of all the problems is that in OO we can encode fixpoints.

But.. all the good about OO is about open fixpoints!
(co-)recursive functions on open data structures

- But with those we can go in loop in hard to predict ways, where there is no while/for and no explicit recursion.
- But with those we can make contracts that become ill founded in hard to predict ways.

Positive type dependencies are everywhere in OO programming

OO patterns are all about inserting interfaces

- to split the graph of static types in layers that can be compiled in separate libraries,
- while allowing the graph of actual type dependencies to be circular.

Positive type dependencies are everywhere in OO programming

But the problem is more fundamental.

Any method that takes a non primitive/non final type can sneak non termination or ill founded contracts in. Consider the following:

- is this well founded?
- is it feasible to check termination since IntList is not final?

```
class IntList{
  /*ensures this.size()+other.size()==this.concat(other).size()*/
  IntList concat(IntList other){ /*..*/ }

  int size(){ /*..*/ }
}
```

Positive type dependencies

Can we simply be safe and prevent Positive type occurrences as Coq does?
Coq, Agda and Lean all restrict the shape of inductively defined data types to avoid positive occurrences of the inductively defined type.

Valid Example:

```
1 Inductive nat : Type :=
2   | O : nat
3   | S : nat → nat. //Given a nat, S produces a nat.
```

Invalid example:

```
1 Inductive nat : Type :=
2   | O : nat
3   | Err : (nat → nat) → nat.
      //Given a function from nat to nat, Err produces a nat.
```

Positive type dependencies

Invalid example:

```
1 Inductive nat : Type :=
2   | O : nat
3   | Err : (nat → nat) → nat.
```

If this was allowed in Coq, we could use it to encode fixpoints and break the termination checker

Consider the list before `interface List{ .. List concat(List other); ..}`
The Coq encoding of this list datatype would be blocked by the positivity checker for the same reason:

```
1 Inductive list : Type :=
2   | List : (List → List) → List.
  //Given an implementation for `method concat', produces a list.
```

Positive type dependencies

What about

```
class List{ List concat(NotAList other){ /*..*/ }  }
```

- What if '`NotAList`' has a List field?
- What if '`Evil extends NotAList`' has a List field?
- What if '`Evil`' simply makes a '`new MyList()`' inside a method call?
- Would this be usable to encode fixpoints/omega?

So, no positivity checker for OO

- Other solutions?
- …

So, no positivity checker for OO

- Other solutions?
- …
- …

So, no positivity checker for OO

- Other solutions?
- …
- …
- …

So, no positivity checker for OO

- Other solutions?

- What if we get rid of the general pre/post condition mindset?

Any Question?

We can now start our group discussion!