Formal Reasoning about Programs and Programming Languages

Amin Timany

Aarhus University,
Aarhus, Denmark

Fri 09 Apr 2021

Introduction

It is important for safety and security that programs are correct, especially in critical applications, *e.g.*, online banking

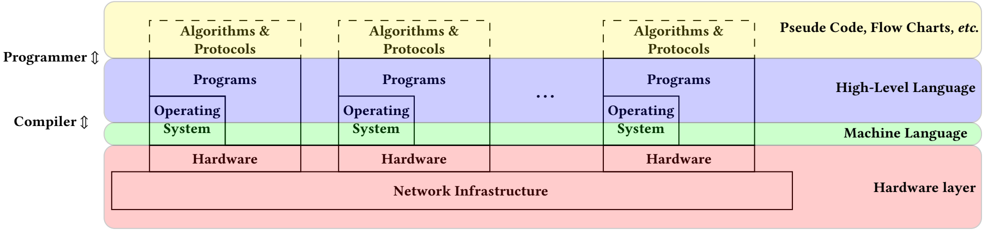
Aim: use formal and mathematical tools to prove correctness of software systems

Methodology:

- Make a mathematical model of the system
- Study the mathematical model using formal logic and mathematical tools

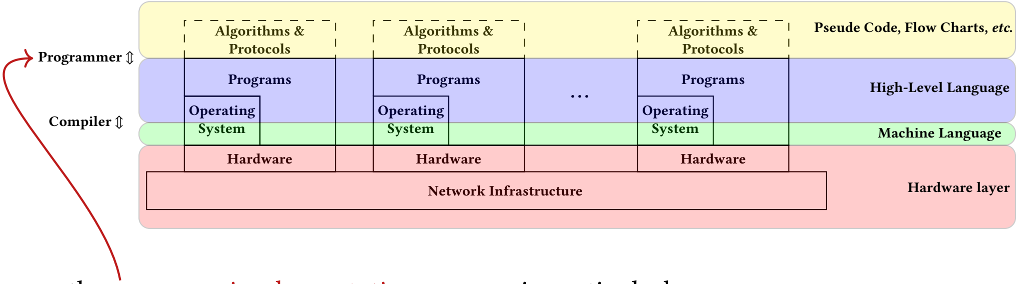
Introduction

There are many levels abstraction consider; all these levels can benefit from formal methods



Introduction

There are many levels abstraction consider; all these levels can benefit from formal methods

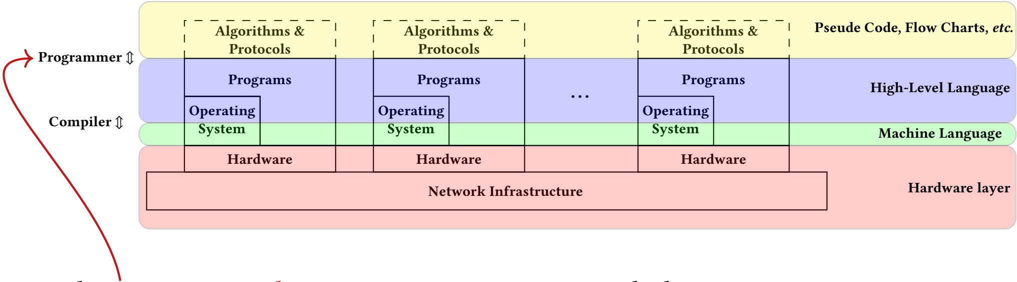


However, the program implementation process is particularly error-prone

- Modern PL features, *e.g.*, concurrency, are challenging to reason about (formally and informally)
- Bugs can introduce serious security vulnerabilities

Introduction

There are many levels abstraction consider; all these levels can benefit from formal methods



However, the program implementation process is particularly error-prone

- Modern PL features, *e.g.*, concurrency, are challenging to reason about (formally and informally)
- Bugs can introduce serious security vulnerabilities

My research focuses on formal verification of programs and programming languages

Example of Security Vulnerability in Implementation: Heartbleed

A bug in OpenSSL's implementation of the heartbeat feature:

- One side sends a heartbeat request message m together with a number l
- The other side sends the first l characters of m back to signal that it is alive

Example of Security Vulnerability in Implementation: Heartbleed

A bug in OpenSSL's implementation of the heartbeat feature:

- One side sends a heartbeat request message m together with a number l
- The other side sends the first l characters of m back to signal that it is alive

A simplified version implementation:

```
void answer_heartbeat(SSL *req, unsigned int l){
  send_reply(l, req->data);
}
```

Example of Security Vulnerability in Implementation: Heartbleed

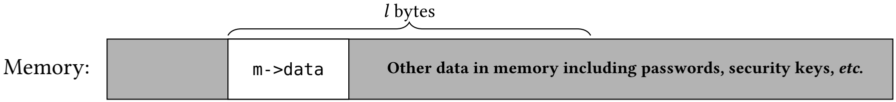
A bug in OpenSSL's implementation of the heartbeat feature:

- One side sends a heartbeat request message m together with a number l
- The other side sends the first l characters of m back to signal that it is alive

A simplified version implementation:

```
void answer_heartbeat(SSL *req, unsigned int l){
  send_reply(l, req->data);
}
```

What happens if $l > length(m)$?



l bytes

Memory:

`m->data`

Other data in memory including passwords, security keys, *etc.*

Example of Security Vulnerability in Implementation: Heartbleed

A bug in OpenSSL’s implementation of the heartbeat feature:

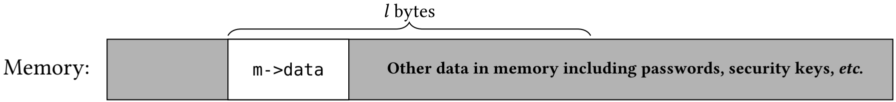
- One side sends a heartbeat request message m together with a number l
- The other side sends the first l characters of m back to signal that it is alive

A simplified version implementation:

```
void answer_heartbeat(SSL *req, unsigned int l){
  send_reply(l, req->data);
}
```

What happens if $l > length(m)$?

This is a *memory violation* and **would have been caught** had the program been verified.



Example of Security Vulnerability in Implementation: Heartbleed

A bug in OpenSSL's implementation of the heartbeat feature:

- One side sends a heartbeat request message m together with a number l
- The other side sends the first l characters of m back to signal that it is alive

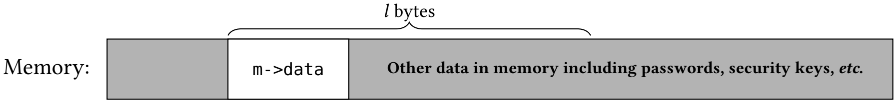
A simplified version implementation:

```
void answer_heartbeat(SSL *req, unsigned int l){
  if(l > req->length){return;}
  send_reply(l, req->data);
}
```

The fix

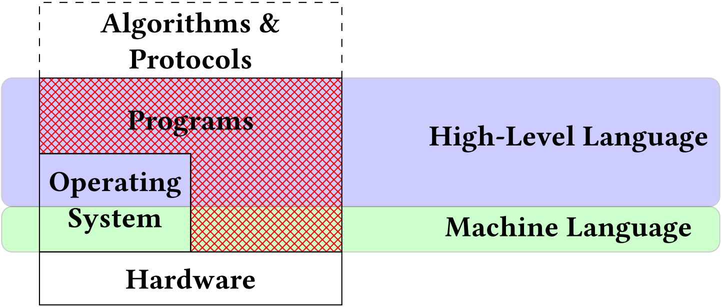
What happens if $l > length(m)$?

This is a *memory violation* and **would have been caught** had the program been verified.



My Research Focus

In my research I focus on reasoning about programs and programming languages



For this purpose I use:

- Formal and mathematical logic: program logics (*the Iris framework*)
- Proof assistants (Coq) to mechanize results (machine-checked mathematical proofs)

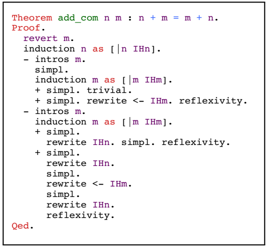
The Proof Assistant Coq

A proof assistant based on the Calculus of Inductive Constructions

- Coq is itself a programming language:
  - Curry-Howard correspondence (types are theorems programs are proofs)
  - It has an interesting meta-theory called *type theory*
- Proofs written and checked against foundational mathematical principles:
  - Coq only understands functions and the concept of induction

An example:

- Commutativity of addition for natural numbers (proven together with Pre-Talent track students)
- Proof automation can help but still this demonstrates the level of formality



```
Theorem add_com n m : n + m = m + n.
Proof.
  revert m.
  induction n as [|n IHn].
  - intros m.
    simpl.
    induction m as [|m IHm].
    + simpl. trivial.
    + simpl. rewrite <- IHm. reflexivity.
  - intros m.
    induction m as [|m IHm].
    + simpl.
      rewrite IHn. simpl. reflexivity.
    + simpl.
      rewrite IHn.
      simpl.
      rewrite <- IHm.
      simpl.
      rewrite IHn.
      reflexivity.
Qed.
```

Proof assistants are the highest standard of rigor for mathematical proofs

The Proof Assistant Coq

We use Coq to reason about state-of-the-art programs and programming languages:

- We define the precise mathematical model of program execution
- The level of details in these models necessitates the use of proof assistants and program logics
- We define program logics (*the Iris framework*) for these programs
- Use these to prove correctness of programs

This is the sate-of-the-art of research in program verification published at the top international conferences, *e.g.*, POPL, ESOP, ICFP

In this talk:

- How we achieve this
- Examples of recent work in this area

How is this feasible?

How can we reason about the state-of-the-art programs at this level of details?

How is this feasible?

How can we reason about the state-of-the-art programs at this level of details?

Modular Proofs!

Modular Proofs and Modular Reasoning about Programs

Curry-Howard correspondence: types are theorems programs are proofs

Software Engineering:

To develop and maintain large programs:

- Divide the program into modules: functions, classes, *etc.*
- Libraries: data structures, networking, GUI, *etc.*

⋮

Proof Engineering:

To develop and maintain large programs:

- Divide the proof into modules: theorems, lemmas, *etc.*
- Libraries: arithmetic, finite sets, *etc.*

⋮

Modular Proofs and Modular Reasoning about Programs

Curry-Howard correspondence: types are theorems programs are proofs

Software Engineering:

To develop and maintain large programs:

- Divide the program into modules: functions, classes, *etc.*
- Libraries: data structures, networking, GUI, *etc.*

  ⋮

Proof Engineering:

To develop and maintain large programs:

- Divide the proof into modules: theorems, lemmas, *etc.*
- Libraries: arithmetic, finite sets, *etc.*

  ⋮

Hence, our program logic supports modular reasoning about realistic effectful programs:

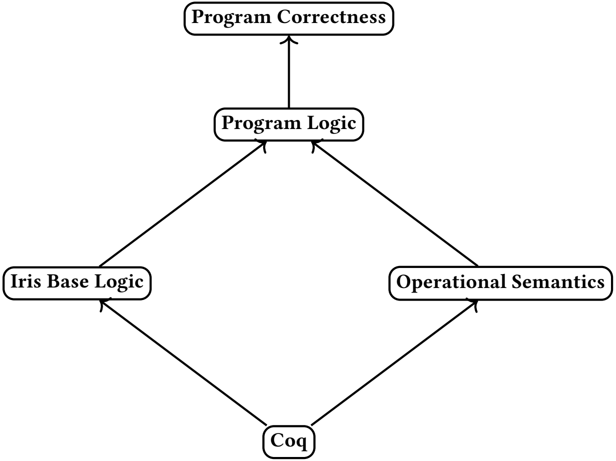
Modular reasoning with respect to program modules

- We reason about each module in isolation and compose those proofs

What is Iris?

A Modular Framework for Constring Program Logics

→ : Built on top of



Program Correctness

Program Logic

Iris Base Logic

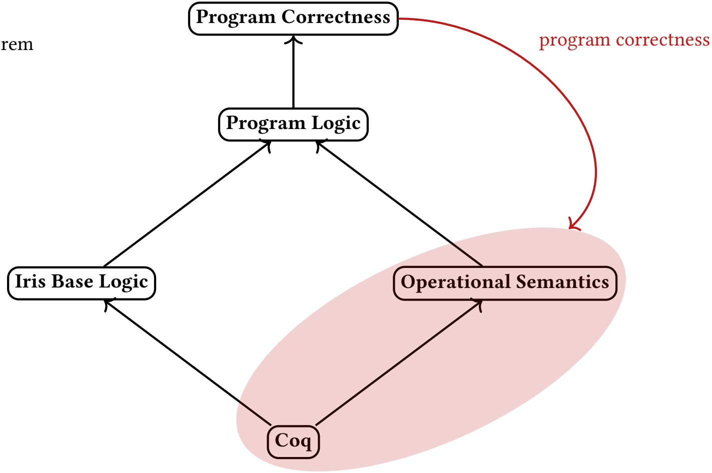
Operational Semantics

Coq

What is Iris?

A Modular Framework for Constring Program Logics

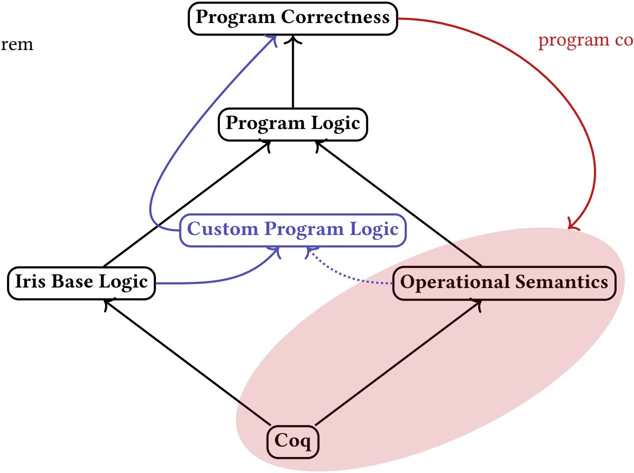
→ : Built on top of
→ : Iris's adequacy theorem



What is Iris?

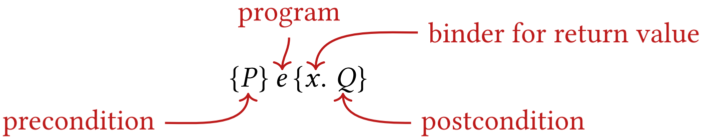
A Modular Framework for Constring Program Logics

→ : Built on top of
→ : Iris's adequacy theorem
□ : User-defined



Program Logic

A Hoare-style logic:



program

binder for return value

$$\{P\}\ e\ \{x.\ Q\}$$

precondition

postcondition

Examples:

$\{\text{True}\}$	$\{\text{isCounter}(c, n)\}$	$\{\text{isCounter}(c, n)\}$
$\quad\text{newCounter}\ ()$	$\quad\text{incr}\ c$	$\quad\text{read}\ c$
$\{x.\ \text{isCounter}(x, 0)\}$	$\{x.\ x = () * \text{isCounter}(c, n + 1)\}$	$\{x.\ x = n * \text{isCounter}(c, n)\}$

Preconditions, postconditions, and invariants[1] allow us to specify conditions for *other* modules.

---

[1] Not presented in this talk

Adequacy Theorem

Theorem (Adequacy)

If we prove

$$\{\mathit{True}\}\; e\; \{x.\ \phi(x)\}$$

in Iris*, then e is safe (e.g., no memory violations) and we have* $\phi(v)$ *for the computed value v.*

Note: this rules out Heartbleed

Example of Modular Reasoning: Function Calls

$\{\text{True}\}$

```
    let c = newCounter () in
    incr c;
    incr c;
    read c
```

$\{x.\ x = 2\}$

Example of Modular Reasoning: Function Calls

$\{\text{True}\}$

$\quad$ `let` $c = \text{newCounter}\ ()$ `in`

$\{\text{isCounter}(c, 0)\}$

$\quad \text{incr}\ c;$

$\{\text{isCounter}(c, 1)\}$

$\quad \text{incr}\ c;$

$\{\text{isCounter}(c, 2)\}$

$\quad \text{read}\ c$

$\{x.\ x = 2 * \text{isCounter}(c, 2)\}$

$\{x.\ x = 2\}$

No need to look at the implementations of newCounter, incr, or read, we just look at the specs.

Example of Modular Reasoning: Concurrency

The parallel composition of two programs e_1 and e_2 (written $e_1||e_2$):

- Runs e_1 and e_2 concurrently in two different threads
- Returns a pair of values (v_1, v_2) corresponding to e_1 and e_2 respectively
- The two programs may work on *shared memory*
- The semantics depends on the order of thread scheduling

The following Hoare-par rule enables modular reasoning about parallel composition:

$$\textsc{Hoare-par}\quad \frac{\{P_1\}\ e_1\ \{x.\ \phi_1(x)\} \qquad \{P_2\}\ e_2\ \{x.\ \phi_2(x)\}}{\{P_1 * P_2\}\ e_1||e_2\ \{x.\ x = (v_1, v_2) * \phi_1(v_1) * \phi_2(v_2)\}}$$

Let’s see a few examples of recent works in this area by my collaborators and I

Reasoning about Distributed Systems (ESOP'20)

Efficient implementations often use advanced features like node-local concurrency and higher-order memory

It is well known that reasoning about distributed programs is difficult

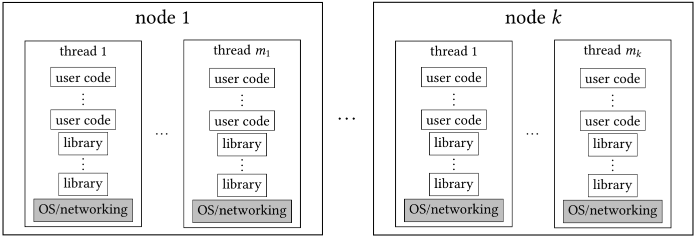
Traditionally, most works focus on verifying a high-level model of the system

We introduced Aneris: a program logic for modular verification of distributed systems

Reasoning about Distributed Systems (ESOP’20)

Modular reasoning about distributed systems:

- Horizontal modularity: nodes, and threads, are verified in isolation and the proofs are composed
- Vertical modularity: library code is verified separately and library clients are verified against the library specs



Reasoning about Causal Consistency (POPL'21)

According to the **CAP** theorem a distributed database cannot satisfy all of the following:

- Consistency: we always read the latest data
- Availability: every request is responded to
- Partition tolerance: system still functions if some of the replicas fail

Causally consistent databases:

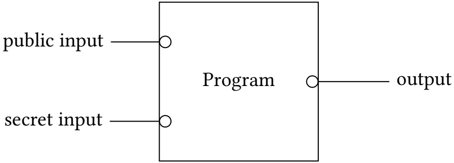
- Sacrifice consistency in favor of availability and partition tolerance
- Even if we don't receive the latest data, we never receive data out of causal order:
    - Example of violation of causal order: receiving response to a message in a group onversation before the message itself

We developed a novel specification (in Aneris) for causally consistent databases, which used to:

- Verify an implementation of a causally consistent database
- Prove correctness of clients of the database (similar to the counter example earlier)

Reasoning about Non-interference (POPL'21)

Non-interference (a security property): output does not leak the secret



A common approach: tracking the level of secrecy of data in the type system

- Each type is annotated with a level, *e.g.*, bool^H, bool^L
- The type system ensures that no data flows from high inputs to low outputs
- Non-interference (termination in-sensitive) in terms of types:

$$\text{TINI:} \qquad \text{for any function } f : \mathsf{bool}^H \to \mathsf{bool}^L \text{ we have } f\ \texttt{true} \approx f\ \texttt{false}$$

Reasoning about Non-interference (POPL'21)

We proved termination-insensitive non-interference:

- For the most advanced type system to date
- Required a novel program logic
- We can reason about both well-typed code and ill-typed code

We do this as follows:

- We define a program logic for termination-insensitive reasoning
- We use it to express non-interference properties of programs of each type such that:

$$\llbracket \mathrm{bool}^H \longrightarrow \mathrm{bool}^L \rrbracket(f) \ \text{ implies } \ f\ \texttt{true} \approx f\ \texttt{false}$$

- We prove that any well-typed program $e : \tau$ we have $\llbracket \tau \rrbracket(e)$
- Hence, the TINI property holds

Other Examples

There are other interesting examples that I did not cover in this talk, *e.g.*,

- Reasoning about machine code (assembly) of so-called capability machines (POPL'21)
- Studying gradual type systems (POPL'21)
- Reasoning about atomicity of advanced concurrent programs (POPL'20)
- Reasoing about continuations (ICFP'19)
- Properties of the ST-monad (POPL'18)
- *etc.*

If you are interested, you can find the full list of my publications at:
`https://cs.au.dk/~timany/publications`

Thanks