# CS6217: Topics in Programming Languages & Software Engineering

Ilya Sergey

*Introduction*

ilyasergey.net/CS6217

# Why taking this class?

Our goal:
make sure that
software behaves correctly.

# Conventional Methods

## Test

- Run the program on well-chosen inputs.
- Compare observed behaviours with expected behaviours.

## Review

- Carefully proofread the code, the tests, the design documents, . . .

## Code Analysis

- Mathematical study of some aspects of the program: numerical precision, time or space complexity, etc.
- Pencil and paper, or with machine assistance (static analysis tools).

slide content credit: Xavier Leroy

# Limitations of Testing

*Testing shows the presence, not the absence of bugs.*
(E. W. Dijkstra, 1969)

We test a small number of all possible behaviours of the program.
Some bugs trigger very rarely!

**Example (carry propagation in a cryptographic library)**

Add 2 `* ta * tb` to `c2:c1:c0` while "optimizing" carry propagation.

```
BN_UMULT_LOHI(t0,t1,ta,tb);
t2 = t1+t1; c2 += (t2<t1)?1:0;
t1 = t0+t0; t2 += (t1<t0)?1:0;
c0 += t1; t2 += (c0<t1)?1:0;
c1 += t2; c2 += (c1<t2)?1:0;
```

# Limitations of Code Review

*Given enough eyeballs, all bugs are shallow.*
(Eric Raymond, 1999)

Reviewers are tired or distracted.

Some codes such as hot fixes are not reviewed much.

**Example (the `goto fail` bug)**

```
if ((err=SSLHashSHA1.update(&hashCtx,&signedParams)) != 0)
    goto fail;
    goto fail;
if ...
...
fail: return err;
```

# Limitations of Code Analysis

*Beware of bugs in the above code; I have only proved it correct, not tried it.*
(Donald E. Knuth, 1977)

Risk of errors in pencil-and-paper analyses and of unsoundness in static analysis tools.

Possible gap between the analysis and the actual program or its actual execution context.

**Example (Ariane 501)**

Overflow in a conversion 64-bit FP number $\to$ 16-bit integer.

An analysis conducted in the context of Ariane 4 proved that the converted quantity, called BH, always fits in 16 bits. The analysis was invalid in the context of Ariane 5.

# Deductive Verification (*aka* Program Proof)

Logical reasoning that establishes properties that hold for *all possible executions* of the program.

Unlike other "formal methods", the properties established go all the way up to full functional correctness *wrt* a *specification*.

Practical interest:

- Obtaining guarantees *stronger* than those we can get using testing and review.
- Finding bugs we cannot find by other means (e.g., via static analysis).

slide content credit: Xavier Leroy

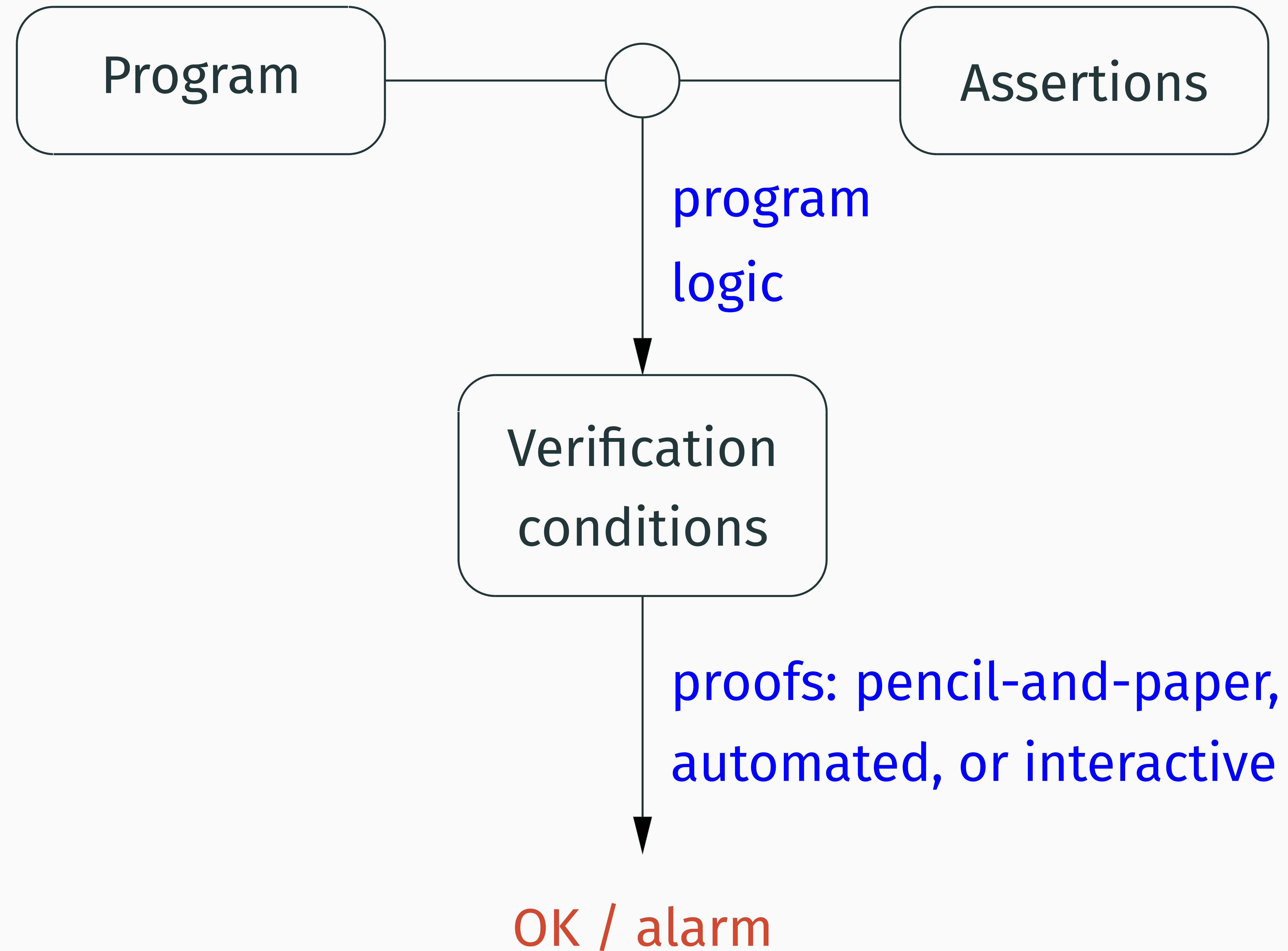
# Program Logics

A program logic provides us with a **specification language** and **reasoning principles** to reason about program behaviours.

Specifications generally consist in **logical assertions** about the program:

- **preconditions**: hypotheses on inputs
  (function parameters; initial values of variables)
- **postconditions**: guarantees on outputs
  (function results; final values of variables)
- **invariants**: guarantees on the states at a program point
  (loop invariants, data structure invariants, . . . )

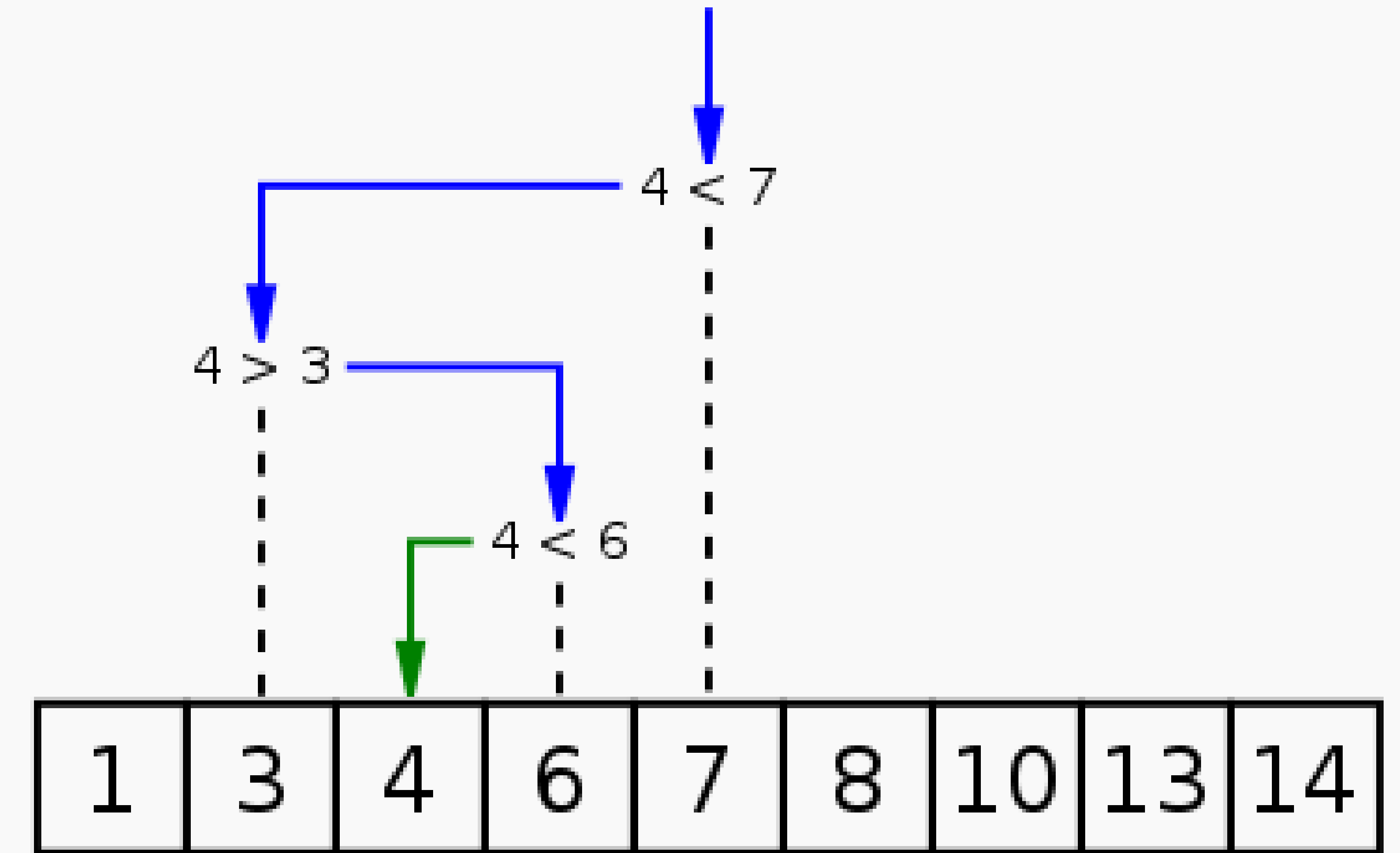
# Program Logics and Deductive Verification




slide content credit: Xavier Leroy

# Hunting for Bugs: Binary Search

# Binary Search




```
l = 0; h = a.length - 1;
while (l <= h) {
  m = (l + h) / 2;
  if (a[m] == v) return m;
  if (a[m] < v) h = m - 1; else l = m + 1;
}
return -1;
```

slide content credit: Xavier Leroy

# Long History...

```
l = 0; h = a.length - 1;
while (l <= h) {
  m = (l + h) / 2;
  if (a[m] == v) return m;
  if (a[m] < v) h = m - 1; else l = m + 1;
}
return -1;
```

1946: John Mauchly, *Moore School Lectures*

1960: Derrick H. Lehmer publishes the modern algorithm

1986: Jon Bentley, *Programming pearls*, chapter 4

2004: Bug report: `java.util.Arrays.binarySearch()` will throw an `ArrayIndexOutOfBoundsException` if the array is large.

2006: Joshua Bloch, *Nearly All Binary Searches and Mergesorts are Broken*.

# The source of the bug: an arithmetic overflow

**m = (l + h) / 2;**

We have **0 ≤ l ≤ h < a.length**.

**l + h** can overflow if **a.length** is large enough.

In Java, **l + h** becomes negative, as well as **m**,
hence **a[m]** raises an “out of bounds” exception.

In C, we have a so-called *undefined behaviour*.
Often, the program continues with the wrong value of **m**.
Worse things can happen (??).

A simple fix: **m = l + (h - l) / 2;**

slide content credit: Xavier Leroy

# This bug is hard to find

## Test

- We rarely test on very big inputs.
- A 64-bit machine and several Gb of RAM are required to trigger this bug.

## Review

- The formula $(l + h)/2$ is so familiar as to raise no suspicion.
- Reviewers are likely to suggest "optimising" $l + (h - l)/2$ as $(l + h)/2$.

## Code Analysis

- A variation interval analysis can detect the problem, but such analyses are considered too slow to run in production.

# Demo

Deductive verification of binary search using the Dafny tool.

https://github.com/cs6217/binary-search-dafny

# Deductive Verification in Academia and Industry

**Separation Logic: A Logic for Shared Mutable Data Structures**

John C. Reynolds*
Computer Science Department
Carnegie Mellon University
john.reynolds@cs.cmu.edu

**Abstract**

*In joint work with Peter O'Hearn and others, based on early ideas of Burstall, we have developed an extension of Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structure.*

*The simple imperative programming language is extended with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation and deallocation of storage. Assertions are extended by introducing a "separating conjunction" that asserts that its subformulas hold for disjoint parts of the heap, and a closely related "separating implication". Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing.*

*In this paper, we will survey the current development of this program logic, including extensions that permit unrestricted address arithmetic, dynamically allocated arrays, and recursive procedures. We will also discuss promising future directions.*

depends upon complex restric
structures. To illustrate this p
its solution, consider a simple
gram performs an in-place reve

j := **nil** ; **while** i ≠ **ni**
(k := [i + 1] ; [i +

(Here the notation $[e]$ denotes
address $e$.)

The invariant of this progra
lists representing two sequenc
flection of the initial value $\alpha_0$
nating the reflection of $\alpha$ onto

$\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}$

where the predicate **list** $\alpha$ i is
length of $\alpha$:

$\mathsf{list}\ \epsilon\ \mathsf{i} \stackrel{\text{def}}{=} \mathsf{i} = \mathbf{nil} \qquad \mathsf{list}(\mathsf{a}\cdot$

Excerpt: Table of Contents
and first three chapters

**PROGRAM LOGICS**
**FOR CERTIFIED COMPILERS**

ANDREW W. APPEL
ROBERT DOCKINS, AQUINAS HOBOR, LENNART BERINGER,
JOSIAH DODDS, GORDON STEWART, SANDRINE BLAZY,
XAVIER LEROY

Be
Compositional

Code
Infer
TAKIPI

# What you will learn in this course

- *Understanding* specifications in terms of program logics
- Understanding *proofs* of program correctness in program logics
- Grasp *new concepts* in program logics and formal reasoning
- Design *invariants* for program verification
- (Optional) Using *tools* for logic-based program verification

# Course Logistics

# Lectures and Presentations

- Weeks 1-7: lectures covering the following topics (tentatively)
  - Floyd-Hoare Style reasoning. Loop invariants. Weakest Precondition Calculus.
  - Separation Logic and reasoning about programs with pointers.
  - Verification of Concurrent Programs.  Concurrent Separation Logic
  - Mechanically verifying OCaml programs with Separation Logic in Coq proof assistant
  - Program logics for fine-grained concurrency and distributed systems
  - Reasoning about Hypersafety properties. Relational Program Logics
  - Over- and Under-approximate reasoning. Incorrectness Logic
- Weeks 8-11: Paper-based presentations and quizzes
- Weeks 12-13: Project presentations

# Paper-based presentations

- Choosing a topic (not a single paper!) and prepare a 45-min talk on it.
- In addition, prepare a *5-question quiz* to test understanding
- How to make good presentations:
  - Provide motivation
  - Include one or several examples
  - Prepare questions for the audience
- More on how to prepare good paper-based talks:
  - "How to give talks that people can follow" by Derek Dreyer
  - https://youtu.be/TCytsY8pdsc

# Research Projects

- Select a “tricky” program you’d like to verify
  - Formulate its correctness specification
  - Come up with a “client” program that makes use of the specification
- Choose a logic to do that in and formalise the full proof
  - Both “paper-and-pencil” and mechanised proofs are acceptable
- Working in teams of one or two
  - For teams of two, I expect a mechanised proof
  - Teams of one can deliver a hand-written proof typeset in LaTeX

# Next in This Lecture

- Formal reasoning about software
- The discovery of program logics