Lecture 1: Introduction

Łukasz Czajka

What is this lecture about?

- Coq

What is this lecture about?

- Coq
  - and applications to program verification

What is this lecture about?

- Coq
  - and applications to program verification
- Or: Coq for (functional) programmers with some background in logic (as taught in a typical bachelor CS program).

Program proof – motivation

Program proof – motivation

- We have all committed errors in our programs.

Program proof – motivation

- We have all committed errors in our programs.
- Some errors are graver than others.

THERAC-25

- Therac-25 was a computer-controlled radiation therapy machine produced in 1982.

THERAC-25

- Therac-25 was a computer-controlled radiation therapy machine produced in 1982.
- A software error caused a massive radiation overdose in at least six incidents. People died.

THERAC-25

- Therac-25 was a computer-controlled radiation therapy machine produced in 1982.
- A software error caused a massive radiation overdose in at least six incidents. People died.
- Cause: a race condition.

THERAC-25

- Therac-25 was a computer-controlled radiation therapy machine produced in 1982.
- A software error caused a massive radiation overdose in at least six incidents. People died.
- Cause: a race condition.

Other critical software bugs in medical equipment: heart devices (2008), infusion pumps (2015, 2019), ...

Ariane 5

- Ariane 5 was a $500 million commercial rocket built by the European Space Agency.

Ariane 5

- Ariane 5 was a $500 million commercial rocket built by the European Space Agency.
- On its first flight in 1996 it exploded about 40 seconds after takeoff.

Ariane 5

- Ariane 5 was a $500 million commercial rocket built by the European Space Agency.
- On its first flight in 1996 it exploded about 40 seconds after takeoff.

- Cause: integer overflow.

Ariane 5

- Ariane 5 was a $500 million commercial rocket built by the European Space Agency.
- On its first flight in 1996 it exploded about 40 seconds after takeoff.

- Cause: integer overflow.

Other critical software bugs in spacecraft: NASA Mars Climate Orbiter (1999), Japanese Hitomi satellite (2016), ...

An old joke from the 1990s

- Q: How many Pentium designers does it take to screw in a light bulb?

An old joke from the 1990s

- Q: How many Pentium designers does it take to screw in a light bulb?
- A: 1.99904274017, but that's close enough for non-technical people.

Pentium FDIV bug

- A bug in floating-point division caused, under certain circumstances, the result to be incorrect beyond the 4th digit.

Pentium FDIV bug

- A bug in floating-point division caused, under certain circumstances, the result to be incorrect beyond the 4th digit.
- Discovered in October 1994.

Pentium FDIV bug

- A bug in floating-point division caused, under certain circumstances, the result to be incorrect beyond the 4th digit.
- Discovered in October 1994.
- In December 1994 Intel offered to replace affected chips upon request.

Pentium FDIV bug

- A bug in floating-point division caused, under certain circumstances, the result to be incorrect beyond the 4th digit.
- Discovered in October 1994.
- In December 1994 Intel offered to replace affected chips upon request.
- Total cost: $457 million.

Program proof – motivation

"Program testing can be used to show the presence of bugs, but never to show their absence!" Edsger W. Dijkstra

Rice's theorem

Theorem (Rice)

Every non-trivial semantic property of programs is undecidable.

Formal methods

- Approximation: static program analysis.

Formal methods

- Approximation: static program analysis.
  - Answer: definitely satisfies a specific property, or maybe not.

Formal methods

- Approximation: static program analysis.
  - Answer: definitely satisfies a specific property, or maybe not.
  - False alarms!

Formal methods

- Approximation: static program analysis.
  - Answer: definitely satisfies a specific property, or maybe not.
  - False alarms!
  - Examples: data-flow analysis, abstract interpretation.

Formal methods

- Approximation: static program analysis.
  - Answer: definitely satisfies a specific property, or maybe not.
  - False alarms!
  - Examples: data-flow analysis, abstract interpretation.
  - Automated, applicable to large programs.

Formal methods

- Approximation: static program analysis.
  - Answer: definitely satisfies a specific property, or maybe not.
  - False alarms!
  - Examples: data-flow analysis, abstract interpretation.
  - Automated, applicable to large programs.
- Abstraction: model checking.

Formal methods

- Approximation: static program analysis.
  - Answer: definitely satisfies a specific property, or maybe not.
  - False alarms!
  - Examples: data-flow analysis, abstract interpretation.
  - Automated, applicable to large programs.
- Abstraction: model checking.
  - Verify a model of the system, not the system itself.

Formal methods

- Approximation: static program analysis.
  - Answer: definitely satisfies a specific property, or maybe not.
  - False alarms!
  - Examples: data-flow analysis, abstract interpretation.
  - Automated, applicable to large programs.
- Abstraction: model checking.
  - Verify a model of the system, not the system itself.
  - Automated (mostly), applicable to medium-sized programs.

Formal methods

- Approximation: static program analysis.
  - Answer: definitely satisfies a specific property, or maybe not.
  - False alarms!
  - Examples: data-flow analysis, abstract interpretation.
  - Automated, applicable to large programs.
- Abstraction: model checking.
  - Verify a model of the system, not the system itself.
  - Automated (mostly), applicable to medium-sized programs.
- **Interaction: proof assistants** (this lecture).

Formal methods

- Approximation: static program analysis.
  - Answer: definitely satisfies a specific property, or maybe not.
  - False alarms!
  - Examples: data-flow analysis, abstract interpretation.
  - Automated, applicable to large programs.
- Abstraction: model checking.
  - Verify a model of the system, not the system itself.
  - Automated (mostly), applicable to medium-sized programs.
- **Interaction: proof assistants** (this lecture).
  - Requires a huge effort for real-world systems.

Formal methods

- Approximation: static program analysis.
  - Answer: definitely satisfies a specific property, or maybe not.
  - False alarms!
  - Examples: data-flow analysis, abstract interpretation.
  - Automated, applicable to large programs.
- Abstraction: model checking.
  - Verify a model of the system, not the system itself.
  - Automated (mostly), applicable to medium-sized programs.
- **Interaction: proof assistants** (this lecture).
  - Requires a huge effort for real-world systems.
  - Applicable to relatively small programs.

Validation vs verification

- **Verification**: does the implementation satisfy a given specification? Do we build the thing right?

Validation vs verification

- **Verification**: does the implementation satisfy a given specification? Do we build the thing right?
  - A full machine-checked formal proof can certify (beyond any reasonable doubt) that this is indeed the case.

Validation vs verification

- **Verification**: does the implementation satisfy a given specification? Do we build the thing right?
  - A full machine-checked formal proof can certify (beyond any reasonable doubt) that this is indeed the case.
- **Validation**: is the specification right? Do we build the right thing?

Validation vs verification

- **Verification**: does the implementation satisfy a given specification? Do we build the thing right?
  - A full machine-checked formal proof can certify (beyond any reasonable doubt) that this is indeed the case.
- **Validation**: is the specification right? Do we build the right thing?
  - Formal proof does not directly help.

Validation vs verification

- **Verification**: does the implementation satisfy a given specification? Do we build the thing right?
  - A full machine-checked formal proof can certify (beyond any reasonable doubt) that this is indeed the case.
- **Validation**: is the specification right? Do we build the right thing?
  - Formal proof does not directly help.
  - BUT: writing a formal specification and proving the program correct with respect to it forces you to think more thoroughly about the specification.

A machine-checked formal proof

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

A machine-checked formal proof

- "Beware of bugs in the above code; I have only proved it correct, not tried it." Donald E. Knuth, 1977
  - An out-of-context quote often overused to claim that "formal methods don't work".

A machine-checked formal proof

- "Beware of bugs in the above code; I have only proved it correct, not tried it." Donald E. Knuth, 1977
  - An out-of-context quote often overused to claim that "formal methods don't work".
- A reply: "And what else would you expect if you just wrote your proof on a piece of paper?"

A machine-checked formal proof

- "Beware of bugs in the above code; I have only proved it correct, not tried it." Donald E. Knuth, 1977
  - An out-of-context quote often overused to claim that "formal methods don't work".
- A reply: "And what else would you expect if you just wrote your proof on a piece of paper?"
  - Software correctness proofs vs proofs of mathematical theorems.

A machine-checked formal proof

- "Beware of bugs in the above code; I have only proved it correct, not tried it." Donald E. Knuth, 1977
  - An out-of-context quote often overused to claim that "formal methods don't work".
- A reply: "And what else would you expect if you just wrote your proof on a piece of paper?"
  - Software correctness proofs vs proofs of mathematical theorems.
- Another reply: "Yes, you also need to run and test it (validation)"

A modern proof assistant

- De Bruijn criterion: a small trusted kernel that checks simple proof objects.

A modern proof assistant

- De Bruijn criterion: a small trusted kernel that checks simple proof objects.
- Everything else (interaction, partial type inference, proof and tactic languages, decision procedures, ...) is untrusted (outside the kernel), but produces proof objects to be independently checked by the kernel.

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification,

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification, the program representation in the proof assistant corresponds to your program (a relatively small leap of faith),

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification, the program representation in the proof assistant corresponds to your program (a relatively small leap of faith), your compiler is implemented correctly (it isn't, unless it's CompCert or CakeML),

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification, the program representation in the proof assistant corresponds to your program (a relatively small leap of faith), your compiler is implemented correctly (it isn't, unless it's CompCert or CakeML), your operating system is implemented correctly (it isn't, unless it's seL4),

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification, the program representation in the proof assistant corresponds to your program (a relatively small leap of faith), your compiler is implemented correctly (it isn't, unless it's CompCert or CakeML), your operating system is implemented correctly (it isn't, unless it's seL4), the proof assistant kernel is implemented correctly (this can be formally verified!),

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification, the program representation in the proof assistant corresponds to your program (a relatively small leap of faith), your compiler is implemented correctly (it isn't, unless it's CompCert or CakeML), your operating system is implemented correctly (it isn't, unless it's seL4), the proof assistant kernel is implemented correctly (this can be formally verified!), the proof assistant logic is consistent,

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification, the program representation in the proof assistant corresponds to your program (a relatively small leap of faith), your compiler is implemented correctly (it isn't, unless it's CompCert or CakeML), your operating system is implemented correctly (it isn't, unless it's seL4), the proof assistant kernel is implemented correctly (this can be formally verified!), the proof assistant logic is consistent, the hardware is correct,

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification, the program representation in the proof assistant corresponds to your program (a relatively small leap of faith), your compiler is implemented correctly (it isn't, unless it's CompCert or CakeML), your operating system is implemented correctly (it isn't, unless it's seL4), the proof assistant kernel is implemented correctly (this can be formally verified!), the proof assistant logic is consistent, the hardware is correct, cosmic rays didn't make your hardware malfunction when checking the proof,

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification, the program representation in the proof assistant corresponds to your program (a relatively small leap of faith), your compiler is implemented correctly (it isn't, unless it's CompCert or CakeML), your operating system is implemented correctly (it isn't, unless it's seL4), the proof assistant kernel is implemented correctly (this can be formally verified!), the proof assistant logic is consistent, the hardware is correct, cosmic rays didn't make your hardware malfunction when checking the proof, your eyes didn't malfunction looking at the screen,

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification, the program representation in the proof assistant corresponds to your program (a relatively small leap of faith), your compiler is implemented correctly (it isn't, unless it's CompCert or CakeML), your operating system is implemented correctly (it isn't, unless it's seL4), the proof assistant kernel is implemented correctly (this can be formally verified!), the proof assistant logic is consistent, the hardware is correct, cosmic rays didn't make your hardware malfunction when checking the proof, your eyes didn't malfunction looking at the screen, your brain didn't malfunction processing the neural signals form your eyes,

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification, the program representation in the proof assistant corresponds to your program (a relatively small leap of faith), your compiler is implemented correctly (it isn't, unless it's CompCert or CakeML), your operating system is implemented correctly (it isn't, unless it's seL4), the proof assistant kernel is implemented correctly (this can be formally verified!), the proof assistant logic is consistent, the hardware is correct, cosmic rays didn't make your hardware malfunction when checking the proof, your eyes didn't malfunction looking at the screen, your brain didn't malfunction processing the neural signals form your eyes, ...

How reliable is a machine-checked proof?

- If you typed into a proof assistant a program correctness proof and the proof assistant says your proof is OK, then you can be absolutely sure your program will behave according to its specification.
  - Assuming: you correctly formalised the specification, the program representation in the proof assistant corresponds to your program (a relatively small leap of faith), your compiler is implemented correctly (it isn't, unless it's CompCert or CakeML), your operating system is implemented correctly (it isn't, unless it's seL4), the proof assistant kernel is implemented correctly (this can be formally verified!), the proof assistant logic is consistent, the hardware is correct, cosmic rays didn't make your hardware malfunction when checking the proof, your eyes didn't malfunction looking at the screen, your brain didn't malfunction processing the neural signals form your eyes, ...
- In whatever you do, there are (implicit or explicit) assumptions!

How reliable is a machine-checked proof?

From seL4 verified microkernel FAQ:

- Q: Does seL4 have zero bugs?

How reliable is a machine-checked proof?

From seL4 verified microkernel FAQ:

- Q: Does seL4 have zero bugs?
- A: The functional correctness proof states that, if the proof assumptions are met, the seL4 kernel implementation has no deviations from its specification. (. . .)

How reliable is a machine-checked proof?

From seL4 verified microkernel FAQ:

- Q: Does seL4 have zero bugs?
- A: The functional correctness proof states that, if the proof assumptions are met, the seL4 kernel implementation has no deviations from its specification. (. . .)
  There may still be unexpected features in the specification and one or more of the assumptions may not apply. The security properties may be sufficient for what your system needs, but might not. (. . .)

How reliable is a machine-checked proof?

From seL4 verified microkernel FAQ:

- Q: Does seL4 have zero bugs?
- A: The functional correctness proof states that, if the proof assumptions are met, the seL4 kernel implementation has no deviations from its specification. (...)
  There may still be unexpected features in the specification and one or more of the assumptions may not apply. The security properties may be sufficient for what your system needs, but might not. (...)
  So the answer to the question depends on what you understand a bug to be. In the understanding of formal software verification (code implements specification), the answer is yes. In the understanding of a general software user, the answer is potentially, because there may still be hardware bugs or proof assumptions unmet. For high assurance systems, this is not a problem, because analysing hardware and proof assumptions is much easier than analysing a large software system, the same hardware, and test assumptions.

How reliable is a machine-checked proof?

From seL4 verified microkernel FAQ:

- Q: Does seL4 have zero bugs?
- A: The functional correctness proof states that, if the proof assumptions are met, the seL4 kernel implementation has no deviations from its specification. (. . .)
  There may still be unexpected features in the specification and one or more of the assumptions may not apply. The security properties may be sufficient for what your system needs, but might not. (. . .)
  So the answer to the question depends on what you understand a bug to be. In the understanding of formal software verification (code implements specification), the answer is yes. In the understanding of a general software user, the answer is potentially, because there may still be hardware bugs or proof assumptions unmet. For high assurance systems, this is not a problem, because analysing hardware and proof assumptions is much easier than analysing a large software system, the same hardware, and test assumptions.

Source: `https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html#does-sel4-have-zero-bugs`.

How reliable is a machine-checked proof?

Real software systems don't exist in isolation.

How reliable is a machine-checked proof?

Real software systems don't exist in isolation.

- Q: If I run seL4, is my system secure?

How reliable is a machine-checked proof?

Real software systems don't exist in isolation.

- Q: If I run seL4, is my system secure?
- A: Not automatically, no. Security is a question that spans the whole system, including its human parts. An OS kernel, verified or not, does not automatically make a system secure. In fact, any system, no matter how secure, can be used in insecure ways.

How reliable is a machine-checked proof?

Real software systems don't exist in isolation.

- Q: If I run seL4, is my system secure?
- A: Not automatically, no. Security is a question that spans the whole system, including its human parts. An OS kernel, verified or not, does not automatically make a system secure. In fact, any system, no matter how secure, can be used in insecure ways.
  However, if used correctly, seL4 provides the system architect and user with strong mechanisms to implement security policies, backed by specific security theorems.

How reliable is a machine-checked proof?

Real software systems don't exist in isolation.

- Q: If I run seL4, is my system secure?
- A: Not automatically, no. Security is a question that spans the whole system, including its human parts. An OS kernel, verified or not, does not automatically make a system secure. In fact, any system, no matter how secure, can be used in insecure ways.
  However, if used correctly, seL4 provides the system architect and user with strong mechanisms to implement security policies, backed by specific security theorems.

Source: `https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html#if-i-run-sel4-is-my-system-secure`.

How reliable is a machine-checked proof?

The purpose of formal methods is not to provide an "absolute" correctness proof, but to substantially increase reliability.

Common confusion

interactive theorem prover (proof assistant)
$\neq$
(fully) automated theorem prover

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.
- 1960s: pioneering proof checkers.

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.
- 1960s: pioneering proof checkers.
- 1968: Automath project (de Bruijn, Netherlands).

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.
- 1960s: pioneering proof checkers.
- 1968: Automath project (de Bruijn, Netherlands).
  - proof checker based on the Curry-Howard correspondence: formulas as types and proofs as programs,

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.
- 1960s: pioneering proof checkers.
- 1968: Automath project (de Bruijn, Netherlands).
  - proof checker based on the Curry-Howard correspondence: formulas as types and proofs as programs,
  - de Bruijn criterion (simple proof objects checked by a small kernel).

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.
- 1960s: pioneering proof checkers.
- 1968: Automath project (de Bruijn, Netherlands).
  - proof checker based on the Curry-Howard correspondence: formulas as types and proofs as programs,
  - de Bruijn criterion (simple proof objects checked by a small kernel).
- 1972: LCF prover (Robin Milner, Stanford & Edinburgh).

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.
- 1960s: pioneering proof checkers.
- 1968: Automath project (de Bruijn, Netherlands).
  - proof checker based on the Curry-Howard correspondence: formulas as types and proofs as programs,
  - de Bruijn criterion (simple proof objects checked by a small kernel).
- 1972: LCF prover (Robin Milner, Stanford & Edinburgh).
  - based on the Logic of Computable Functions by Dana Scott,

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.
- 1960s: pioneering proof checkers.
- 1968: Automath project (de Bruijn, Netherlands).
  - proof checker based on the Curry-Howard correspondence: formulas as types and proofs as programs,
  - de Bruijn criterion (simple proof objects checked by a small kernel).
- 1972: LCF prover (Robin Milner, Stanford & Edinburgh).
  - based on the Logic of Computable Functions by Dana Scott,
  - no proof objects, but a small kernel implements inference rules and ML's type system ensures these are used correctly.

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.
- 1960s: pioneering proof checkers.
- 1968: Automath project (de Bruijn, Netherlands).
  - proof checker based on the Curry-Howard correspondence: formulas as types and proofs as programs,
  - de Bruijn criterion (simple proof objects checked by a small kernel).
- 1972: LCF prover (Robin Milner, Stanford & Edinburgh).
  - based on the Logic of Computable Functions by Dana Scott,
  - no proof objects, but a small kernel implements inference rules and ML's type system ensures these are used correctly.
- 1970s: Mizar (Andrzej Trybulec, Poland).

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.
- 1960s: pioneering proof checkers.
- 1968: Automath project (de Bruijn, Netherlands).
  - proof checker based on the Curry-Howard correspondence: formulas as types and proofs as programs,
  - de Bruijn criterion (simple proof objects checked by a small kernel).
- 1972: LCF prover (Robin Milner, Stanford & Edinburgh).
  - based on the Logic of Computable Functions by Dana Scott,
  - no proof objects, but a small kernel implements inference rules and ML's type system ensures these are used correctly.
- 1970s: Mizar (Andrzej Trybulec, Poland).
  - based on set theory, intended for mathematics formalisation by mathematicians,

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.
- 1960s: pioneering proof checkers.
- 1968: Automath project (de Bruijn, Netherlands).
  - proof checker based on the Curry-Howard correspondence: formulas as types and proofs as programs,
  - de Bruijn criterion (simple proof objects checked by a small kernel).
- 1972: LCF prover (Robin Milner, Stanford & Edinburgh).
  - based on the Logic of Computable Functions by Dana Scott,
  - no proof objects, but a small kernel implements inference rules and ML's type system ensures these are used correctly.
- 1970s: Mizar (Andrzej Trybulec, Poland).
  - based on set theory, intended for mathematics formalisation by mathematicians,
  - declarative proof style,

Proof assistants timeline

- 1950s: first attempts at automated theorem proving.
- 1960s: pioneering proof checkers.
- 1968: Automath project (de Bruijn, Netherlands).
  - proof checker based on the Curry-Howard correspondence: formulas as types and proofs as programs,
  - de Bruijn criterion (simple proof objects checked by a small kernel).
- 1972: LCF prover (Robin Milner, Stanford & Edinburgh).
  - based on the Logic of Computable Functions by Dana Scott,
  - no proof objects, but a small kernel implements inference rules and ML's type system ensures these are used correctly.
- 1970s: Mizar (Andrzej Trybulec, Poland).
  - based on set theory, intended for mathematics formalisation by mathematicians,
  - declarative proof style,
  - as of today: has one of the largest libraries of formalised "mainstream" mathematics (MML – Mizar Mathematical Library).

Proof assistants timeline

- 1980s: NuPRL (Constable, Cornell).

Proof assistants timeline

- 1980s: NuPRL (Constable, Cornell).
  - based on extensional Martin-Löf type theory, intended for program verification.

Proof assistants timeline

- 1980s: NuPRL (Constable, Cornell).
  - based on extensional Martin-Löf type theory, intended for program verification.
- 1980s: HOL (Gordon, Cambridge).

Proof assistants timeline

- 1980s: NuPRL (Constable, Cornell).
  - based on extensional Martin-Löf type theory, intended for program verification.
- 1980s: HOL (Gordon, Cambridge).
  - based on classical higher-order logic.

Proof assistants timeline

- 1980s: NuPRL (Constable, Cornell).
  - based on extensional Martin-Löf type theory, intended for program verification.
- 1980s: HOL (Gordon, Cambridge).
  - based on classical higher-order logic.
  - LCF-style.

Proof assistants timeline

- 1980s: NuPRL (Constable, Cornell).
  - based on extensional Martin-Löf type theory, intended for program verification.
- 1980s: HOL (Gordon, Cambridge).
  - based on classical higher-order logic.
  - LCF-style.
- 1986: Isabelle (Paulson & Nipkow, Cambridge & TU München).

Proof assistants timeline

- 1980s: NuPRL (Constable, Cornell).
  - based on extensional Martin-Löf type theory, intended for program verification.
- 1980s: HOL (Gordon, Cambridge).
  - based on classical higher-order logic.
  - LCF-style.
- 1986: Isabelle (Paulson & Nipkow, Cambridge & TU München).
  - LCF-style framework for encoding logics,

Proof assistants timeline

- 1980s: NuPRL (Constable, Cornell).
  - based on extensional Martin-Löf type theory, intended for program verification.
- 1980s: HOL (Gordon, Cambridge).
  - based on classical higher-order logic.
  - LCF-style.
- 1986: Isabelle (Paulson & Nipkow, Cambridge & TU München).
  - LCF-style framework for encoding logics,
  - Isabelle/HOL, Isabelle/FOL, Isabelle/ZF, ...

Proof assistants timeline

- 1980s: NuPRL (Constable, Cornell).
  - based on extensional Martin-Löf type theory, intended for program verification.
- 1980s: HOL (Gordon, Cambridge).
  - based on classical higher-order logic.
  - LCF-style.
- 1986: Isabelle (Paulson & Nipkow, Cambridge & TU München).
  - LCF-style framework for encoding logics,
  - Isabelle/HOL, Isabelle/FOL, Isabelle/ZF, ...
- 1989: Coq (Huet & Coquand, France).

Proof assistants timeline

- 1980s: NuPRL (Constable, Cornell).
  - based on extensional Martin-Löf type theory, intended for program verification.
- 1980s: HOL (Gordon, Cambridge).
  - based on classical higher-order logic.
  - LCF-style.
- 1986: Isabelle (Paulson & Nipkow, Cambridge & TU München).
  - LCF-style framework for encoding logics,
  - Isabelle/HOL, Isabelle/FOL, Isabelle/ZF, ...
- 1989: Coq (Huet & Coquand, France).
  - based on Coquand's Calculus of Constructions,

Proof assistants timeline

- 1980s: NuPRL (Constable, Cornell).
  - based on extensional Martin-Löf type theory, intended for program verification.
- 1980s: HOL (Gordon, Cambridge).
  - based on classical higher-order logic.
  - LCF-style.
- 1986: Isabelle (Paulson & Nipkow, Cambridge & TU München).
  - LCF-style framework for encoding logics,
  - Isabelle/HOL, Isabelle/FOL, Isabelle/ZF, ...
- 1989: Coq (Huet & Coquand, France).
  - based on Coquand's Calculus of Constructions,
  - satisfies de Bruijn criterion.

Proof assistants timeline

- 1990: ACL2 (Boyer & Moore & Kaufmann, University of Texas).

Proof assistants timeline

- 1990: ACL2 (Boyer & Moore & Kaufmann, University of Texas).
  - based on Lisp + quantifier-free first-order logic with induction,

Proof assistants timeline

- 1990: ACL2 (Boyer & Moore & Kaufmann, University of Texas).
  - based on Lisp + quantifier-free first-order logic with induction,
  - a more interactive industrial-strength successor to the Nqthm (Boyer-Moore) automated theorem prover.

Proof assistants timeline

- 1990: ACL2 (Boyer & Moore & Kaufmann, University of Texas).
  - based on Lisp + quantifier-free first-order logic with induction,
  - a more interactive industrial-strength successor to the Nqthm (Boyer-Moore) automated theorem prover.
- 1992: PVS (SRI International, USA).

Proof assistants timeline

- 1990: ACL2 (Boyer & Moore & Kaufmann, University of Texas).
  - based on Lisp + quantifier-free first-order logic with induction,
  - a more interactive industrial-strength successor to the Nqthm (Boyer-Moore) automated theorem prover.
- 1992: PVS (SRI International, USA).
  - powerful automation, intended for industrial applications,

Proof assistants timeline

- 1990: ACL2 (Boyer & Moore & Kaufmann, University of Texas).
  - based on Lisp + quantifier-free first-order logic with induction,
  - a more interactive industrial-strength successor to the Nqthm (Boyer-Moore) automated theorem prover.
- 1992: PVS (SRI International, USA).
  - powerful automation, intended for industrial applications,
  - no proof objects or small kernel.

Proof assistants timeline

- 1990: ACL2 (Boyer & Moore & Kaufmann, University of Texas).
  - based on Lisp + quantifier-free first-order logic with induction,
  - a more interactive industrial-strength successor to the Nqthm (Boyer-Moore) automated theorem prover.
- 1992: PVS (SRI International, USA).
  - powerful automation, intended for industrial applications,
  - no proof objects or small kernel.
- 1998: Intel hires John Harrison (author of HOL Light) to lead the formal verification of floating-point arithmetic in Intel processors.

Proof assistants timeline

- 1990: ACL2 (Boyer & Moore & Kaufmann, University of Texas).
  - based on Lisp + quantifier-free first-order logic with induction,
  - a more interactive industrial-strength successor to the Nqthm (Boyer-Moore) automated theorem prover.
- 1992: PVS (SRI International, USA).
  - powerful automation, intended for industrial applications,
  - no proof objects or small kernel.
- 1998: Intel hires John Harrison (author of HOL Light) to lead the formal verification of floating-point arithmetic in Intel processors.
- 1999: Agda (Norell & Coquand, Sweden).

Proof assistants timeline

- 1990: ACL2 (Boyer & Moore & Kaufmann, University of Texas).
  - based on Lisp + quantifier-free first-order logic with induction,
  - a more interactive industrial-strength successor to the Nqthm (Boyer-Moore) automated theorem prover.
- 1992: PVS (SRI International, USA).
  - powerful automation, intended for industrial applications,
  - no proof objects or small kernel.
- 1998: Intel hires John Harrison (author of HOL Light) to lead the formal verification of floating-point arithmetic in Intel processors.
- 1999: Agda (Norell & Coquand, Sweden).
  - dependently typed functional programming language based on Martin-Löf type theory.

Proof assistants timeline

- 2000s: Increasing use of proof assistants in hardware verification.

Proof assistants timeline

- 2000s: Increasing use of proof assistants in hardware verification.
- 2005: Nqthm/ACL2 receives the ACM Software Systems Award.

Proof assistants timeline

- 2000s: Increasing use of proof assistants in hardware verification.
- 2005: Nqthm/ACL2 receives the ACM Software Systems Award.
- 2007: Idris (Brady, UK).

Proof assistants timeline

- 2000s: Increasing use of proof assistants in hardware verification.
- 2005: Nqthm/ACL2 receives the ACM Software Systems Award.
- 2007: Idris (Brady, UK).
  - general-purpose dependently typed functional programming language with effects.

Proof assistants timeline

- 2000s: Increasing use of proof assistants in hardware verification.
- 2005: Nqthm/ACL2 receives the ACM Software Systems Award.
- 2007: Idris (Brady, UK).
  - general-purpose dependently typed functional programming language with effects.
- 2008: formal proof (in Coq) of the Four Color Theorem (Gonthier et al, France).

Proof assistants timeline

- 2000s: Increasing use of proof assistants in hardware verification.
- 2005: Nqthm/ACL2 receives the ACM Software Systems Award.
- 2007: Idris (Brady, UK).
  - general-purpose dependently typed functional programming language with effects.
- 2008: formal proof (in Coq) of the Four Color Theorem (Gonthier et al, France).
- 2008: CompCert – a verified (with Coq) optimising compiler for Clight (Leroy et al, France).

Proof assistants timeline

- 2000s: Increasing use of proof assistants in hardware verification.
- 2005: Nqthm/ACL2 receives the ACM Software Systems Award.
- 2007: Idris (Brady, UK).
  - general-purpose dependently typed functional programming language with effects.
- 2008: formal proof (in Coq) of the Four Color Theorem (Gonthier et al, France).
- 2008: CompCert – a verified (with Coq) optimising compiler for Clight (Leroy et al, France).
- 2009: seL4 - a verified (with Isabelle/HOL) OS kernel (Data61, Australia).

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.
- 2014: formal proof (in Isabelle/HOL and HOL Light) of the Kepler conjecture (Hales et al).

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.
- 2014: formal proof (in Isabelle/HOL and HOL Light) of the Kepler conjecture (Hales et al).
- 2014: CakeML – a verified (with Isabelle/HOL) compiler for ML (Data61, Australia).

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.
- 2014: formal proof (in Isabelle/HOL and HOL Light) of the Kepler conjecture (Hales et al).
- 2014: CakeML – a verified (with Isabelle/HOL) compiler for ML (Data61, Australia).
- 2015: F* (Microsoft Research & Inria).

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.
- 2014: formal proof (in Isabelle/HOL and HOL Light) of the Kepler conjecture (Hales et al).
- 2014: CakeML – a verified (with Isabelle/HOL) compiler for ML (Data61, Australia).
- 2015: F* (Microsoft Research & Inria).
  - general-purpose functional programming language with effects aimed at program verification,

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.
- 2014: formal proof (in Isabelle/HOL and HOL Light) of the Kepler conjecture (Hales et al).
- 2014: CakeML – a verified (with Isabelle/HOL) compiler for ML (Data61, Australia).
- 2015: F* (Microsoft Research & Inria).
  - general-purpose functional programming language with effects aimed at program verification,
  - combines automated and interactive proofs.

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.
- 2014: formal proof (in Isabelle/HOL and HOL Light) of the Kepler conjecture (Hales et al).
- 2014: CakeML – a verified (with Isabelle/HOL) compiler for ML (Data61, Australia).
- 2015: F* (Microsoft Research & Inria).
  - general-purpose functional programming language with effects aimed at program verification,
  - combines automated and interactive proofs.
- 2016: mC2 (CertiKOS) – a verified concurrent OS kernel with fine-grained locking (Yale University).

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.
- 2014: formal proof (in Isabelle/HOL and HOL Light) of the Kepler conjecture (Hales et al).
- 2014: CakeML – a verified (with Isabelle/HOL) compiler for ML (Data61, Australia).
- 2015: F* (Microsoft Research & Inria).
  - general-purpose functional programming language with effects aimed at program verification,
  - combines automated and interactive proofs.
- 2016: mC2 (CertiKOS) – a verified concurrent OS kernel with fine-grained locking (Yale University).
- 2018: Amazon Web Services hires John Harrison.

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.
- 2014: formal proof (in Isabelle/HOL and HOL Light) of the Kepler conjecture (Hales et al).
- 2014: CakeML – a verified (with Isabelle/HOL) compiler for ML (Data61, Australia).
- 2015: F* (Microsoft Research & Inria).
  - general-purpose functional programming language with effects aimed at program verification,
  - combines automated and interactive proofs.
- 2016: mC2 (CertiKOS) – a verified concurrent OS kernel with fine-grained locking (Yale University).
- 2018: Amazon Web Services hires John Harrison.
- 2017-202?: Project Everest (Microsoft Research & Inria).

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.
- 2014: formal proof (in Isabelle/HOL and HOL Light) of the Kepler conjecture (Hales et al).
- 2014: CakeML – a verified (with Isabelle/HOL) compiler for ML (Data61, Australia).
- 2015: F* (Microsoft Research & Inria).
  - general-purpose functional programming language with effects aimed at program verification,
  - combines automated and interactive proofs.
- 2016: mC2 (CertiKOS) – a verified concurrent OS kernel with fine-grained locking (Yale University).
- 2018: Amazon Web Services hires John Harrison.
- 2017-202?: Project Everest (Microsoft Research & Inria).
  - work in progress to create a verified HTTPS stack.

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.
- 2014: formal proof (in Isabelle/HOL and HOL Light) of the Kepler conjecture (Hales et al).
- 2014: CakeML – a verified (with Isabelle/HOL) compiler for ML (Data61, Australia).
- 2015: F* (Microsoft Research & Inria).
  - general-purpose functional programming language with effects aimed at program verification,
  - combines automated and interactive proofs.
- 2016: mC2 (CertiKOS) – a verified concurrent OS kernel with fine-grained locking (Yale University).
- 2018: Amazon Web Services hires John Harrison.
- 2017-202?: Project Everest (Microsoft Research & Inria).
  - work in progress to create a verified HTTPS stack.
- 2021: Lean4 – a dependently typed programming language and theorem prover based on the Calculus of Constructions.

Proof assistants timeline

- 2010s: Increasing use of proof assistants in verification of critical (parts of) embedded software.
- 2013: Coq receives the ACM Software Systems Award.
- 2014: formal proof (in Isabelle/HOL and HOL Light) of the Kepler conjecture (Hales et al).
- 2014: CakeML – a verified (with Isabelle/HOL) compiler for ML (Data61, Australia).
- 2015: F* (Microsoft Research & Inria).
  - general-purpose functional programming language with effects aimed at program verification,
  - combines automated and interactive proofs.
- 2016: mC2 (CertiKOS) – a verified concurrent OS kernel with fine-grained locking (Yale University).
- 2018: Amazon Web Services hires John Harrison.
- 2017-202?: Project Everest (Microsoft Research & Inria).
  - work in progress to create a verified HTTPS stack.
- 2021: Lean4 – a dependently typed programming language and theorem prover based on the Calculus of Constructions.
- 2025: Coq renamed to Rocq.

What to expect from this course?

- Despite recent progress, full formal verification of realistic software still requires a huge effort.

What to expect from this course?

- Despite recent progress, full formal verification of realistic software still requires a huge effort.
- After finishing this course do not expect to be able to verify "real-world" software written in mainstream programming languages.

What to expect from this course?

- Despite recent progress, full formal verification of realistic software still requires a huge effort.
- After finishing this course do not expect to be able to verify "real-world" software written in mainstream programming languages.
- After finishing this course do expect to be able to:
  - write small functional programs in Coq (e.g., list sorting, priority queues, balanced search trees), prove their correctness and extract them to OCaml or Haskell,

What to expect from this course?

- Despite recent progress, full formal verification of realistic software still requires a huge effort.
- After finishing this course do not expect to be able to verify "real-world" software written in mainstream programming languages.
- After finishing this course do expect to be able to:
  - write small functional programs in Coq (e.g., list sorting, priority queues, balanced search trees), prove their correctness and extract them to OCaml or Haskell,
  - understand (some of) the theory behind Coq.