How do we build software?

that works

(and be convinced that it does)
Critical Software

**Individual programs**
- Operating systems
- Network stacks
- Crypto
- Medical devices
- Flight control systems
- Power plants
- Home security
- …

**Programming languages**
- Static type systems
- Data abstraction and modularity
- Security controls
- Compiler correctness
LOGICAL FOUNDATIONS

Q: How do we know something is true?
A: We prove it

Q: How do we know that we have a proof?
A: We need to define what it means for something to be a proof.
A proof is a logical sequence of arguments, starting from some initial assumptions

Q: How do we know that we have a valid sequence of arguments? Can any sequence be a proof? E.g.
All humans are mortal
All Greeks are human
Therefore I am a Greek!
A: No, no, no! We need to think harder about valid ways of reasoning...

Aristotle
384 – 322 BC

Euclid
~300 BC
First we need a language...

- **Gottlob Frege**: a German mathematician who started in geometry but became interested in logic and foundations of arithmetic.

- 1879 Published “Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens” (Concept-Script: A Formal Language for Pure Thought Modeled on that of Arithmetic)
  - First rigorous treatment of functions and quantified variables
  - ⊢ A, ¬A, ∀x.F(x)
  - First notation able to express arbitrarily complicated logical statements

Images in this & following slides taken from Wikipedia.

Formalization of Arithmetic

- 1884: Die Grundlagen der Arithmetik (The Foundations of Arithmetic)
- 1893: Grundgesetze der Arithmetik (Basic Laws of Arithmetic, Vol. 1)
- 1903: Grundgesetze der Arithmetik (Basic Laws of Arithmetic, Vol. 2)
- Frege’s goals:
  - isolate logical principles of inference
  - derive laws of arithmetic from first principles
  - set mathematics on a solid foundation of logic

The plot thickens...

Just as Volume 2 was going to print in 1903, Frege received a letter…
Addendum to Frege's 1903 Book

“Hardly anything more unfortunate can befall a scientific writer than to have one of the foundations of his edifice shaken after the work is finished. This was the position I was placed in by a letter of Mr. Bertrand Russell, just when the printing of this volume was nearing its completion.”

— Frege, 1903

Bertrand Russell

- Russell's paradox:
  1. Set comprehension notation:
     \[ \{ x \mid P(x) \} \] "The set of x such that P(x)"
  2. Let X be the set (of sets) \( \{ Y \mid Y \notin Y \} \).
  3. Ask the logical question:
     Does \( X \notin X \) hold?
  4. Paradox! If \( X \in X \) then \( X \notin X \).
     If \( X \notin X \) then \( X \in X \).

- Frege's language could derive Russell's paradox \( \Rightarrow \) it was inconsistent.
- Frege's logical system could derive anything. Oops(!!)

Bertrand Russell
1872 - 1970
Aftermath of Frege and Russell

- Frege came up with a fix... but it made his logic trivial :( 
- 1908: Russell fixed the inconsistency of Frege's logic by developing a theory of types.
- 1910, 1912, 1913, (revised 1927): Principia Mathematica (Whitehead & Russell)
  - Goal: axioms and rules from which all mathematical truths could be derived.
  - It was a bit unwieldy...

"From this proposition it will follow, when arithmetical addition has been defined, that 1+1=2."
—Volume I, 1st edition, page 379

Logic in the 1930s and 1940s

- 1931: Kurt Gödel's first and second incompleteness theorems.
  - Demonstrated that any consistent formal theory capable of expressing arithmetic cannot be complete.
  -
- 1936: Genzen proves consistency of arithmetic.
- 1936: Church introduces the \( \lambda \)-calculus.
- 1936: Turing introduces Turing machines
  - Is there a decision procedure for arithmetic?
  - Answer: no, it's undecidable
  - The famous "halting problem"
    - N.B.: Only in 1938 did Turing get his Ph.D.
- 1940: Church introduces the simple theory of types
Fast Forward…

• Two logicians in 1958 (Haskell Curry) and 1969 (William Howard) observe a remarkable correspondence:

  - types ~ propositions
  - programs ~ proofs
  - computation ~ simplification

• 1967 – 1980’s: N.G. de Bruijn runs Automath project
  – uses the Curry-Howard correspondence for computer-verified mathematics

• 1971: Jean-Yves Girard introduces System F
• 1972: Girard introduces $\lambda w$
• 1972: Per Martin-Löf introduces intuitionistic type theory
• 1974: John Reynolds independently discovers System F

... to the Present

• 1984: Coquand and Huet first begin implementing a new theorem prover “Coq”
• 1985: Coquand introduces the calculus of constructions
  – combines features from intuitionistic type theory and $\lambda w$
• 1989: Coquand and Paulin extend CoC to the calculus of inductive constructions
  – adds “inductive types” as a primitive
• 1992: Coq ported to Xavier Leroy’s OCaml
• 1990’s: up to Coq version 6.2
• 2000-2015: up to Coq version 8.4
• 2017: Coq version 8.6
• 2013: Coq receives ACM Software System Award

So much for foundations… what about the “software” part?

Building Reliable Software

• Suppose you work at (or run) a software company.

• Suppose, like Frege, you’ve sunk 30+ person-years into developing the “next big thing”:
  - Boeing Dreamliner2 flight controller
  - Autonomous vehicle control software for Nissan
  - Gene therapy DNA tailoring algorithms
  - Super-efficient green-energy power grid controller

• Suppose, like Frege, your company has invested a lot of material resources that are also at stake.

• How do you avoid getting a letter like the one from Russell?

  Or, worse yet, not getting the letter, with disastrous consequences down the road?
Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming

- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking

- Technological
  - "lint" tools, static analysis
  - Fuzzers, random testing

- Mathematical
  - Sound type systems
  - Formal verification

More “formal”: eliminate with certainty as many problems as possible.

Less “formal”: Lightweight, inexpensive techniques (that may miss problems)

This isn’t a tradeoff… all of these methods should be used.

Even the most “formal” argument can still have holes:
- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth: “Beware of bugs in the above code; I have only proved it correct, not tried it.”

Can formal methods scale?

Use of formal methods to verify full-scale software systems is a hot research topic!

- **CompCert** – fully verified C compiler
  - Leroy, INRIA

- **Vellvm** – formalized LLVM IR
  - Zdancewic, Penn

- **Verified Software Toolchain**
  - Appel, Princeton

- **Bedrock** – web programming, packet filters
  - Chipala, MIT

- **CertiKOS** – certified OS kernel
  - Shao, Yale
Does it work?

Finding and Understanding Bugs in C Compilers [Yang et al. PLDI 2011]

Random test-case generation

Source Programs

Verifying Compiler: CompCert [Leroy et al.]
<10 bugs found in (at the time unverified) front-end component

79 bugs:
25 critical

202 bugs

325 bugs in total

(8 other C compilers)

Regehr’s Group Concludes

The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.