Administrivia

- Instructor: Steve Zdancewic
  Office hours: Wednesday 3:30-5:00 & by appointment
  Levine 511

- TAs:
  - Dmitri Garbuzov
    Office hours: TBA
  - Jennifer Paykin
    Office hours: TBA

- E-mail: cis500@seas.upenn.edu
- Web site: http://www.seas.upenn.edu/~cis500
- Canvas: https://upenn.instructure.com
- Piazza: http://piazza.com/upenn/fall2013/cis500
Resources

• Course textbook: *Software Foundations*
  – Electronic edition tailor-made for this class
  – Use the version available from the cis500 course web pages.

• Additional books:
  – *Types and Programming Languages* (Pierce, 2002 MIT Press)
  – *Interactive Theorem Proving and Program Development* (Bertot and Castéran, 2004 Springer)
  – *Certified Programming with Dependent Types* (Chlipala, electronic edition)
“Regular” vs. “Advanced” Tracks

• “Advanced” track:
  – More and harder exercises
  – More challenging exams.
  – It is a superset of the “regular” material.

• All students start in the advanced track by default.
• Students who wish to take CIS 500 for WPE I credit (Ph.D.) must take the advanced track.
• Students may switch from advanced to regular track at any time.
  – Notify the course staff.
  – The change is permanent after the first midterm.
• Students wishing to switch (back) to the advanced track:
  – Must do so before the first midterm exam.
  – Must make up all the advanced exercises (or accept the grade penalty).
• Only students taking the advanced track are eligible for an A+. 
Course Policies

• Prerequisites:
  – Undergraduate programming languages or compiler class
  – Significant programming experience
  – Mathematical sophistication

Grading:
• 24% Homework ~12 weekly assignments
• 18% Midterm 1 (tentatively) Oct. 1\textsuperscript{st}
• 18% Midterm 2 (tentatively) Nov. 7\textsuperscript{th}
• 36% Final TBA
• 4% Class participation

⇒ Lecture attendance is crucial!

“Regular” and “Advanced” track students will be graded separately.
Participation Policy

• Class attendance is mandatory.

• We will be using “clickers” for
  – in-class mini quizzes
  – in-class polls about course material

• Clicker use will be your attendance record.

• For next time: buy a clicker at the bookstore.
Homework Policies

• Homework is to be done *individually*.
• Homework must be *submitted via Canvas*
• Homework that is late is subject to:
  – 25% penalty for 1 day late
  – 50% penalty for 2 days late
  – 75% penalty for 3 days late

• Homework is due at *8:00pm* on the due date (generally Thurs.).

• Advanced track students must complete (or try to complete) all non-optional exercises.
  – Missing “advanced” exercises will count against your score.
• Regular track students must complete (or try to complete) all non-optional exercises except those marked “advanced”.
  – Missing “advanced” exercises will not count against your score.
  – (But may help in your understanding of the material)
SOFTWARE FOUNDATIONS

Images in the following slides taken from Wikipedia.
The Story Begins...

- **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
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…
Bertrand Russell

- **Russell’s paradox:**

1. Set comprehension notation:
   \[
   \{ x \mid P(x) \} \quad \text{"The set of } x \text{ such that } P(x)\"
   \]

2. Let \( X \) be the set \( \{ Y \mid Y \notin X \} \).

3. Ask the logical question:
   Does \( X \in X \) hold?

4. Paradox! If \( X \in X \) then \( X \notin X \).
   
   If \( X \notin X \) then \( X \in X \).

- Russell’s paradox destroyed Frege’s logical foundations…

---

CIS 500: Software Foundations
Fall 2013

Zdancewic
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
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 λ-calculus.
• **1936**: Turing introduces Turing machines
  – Is there a decision procedure for arithmetic?
  – Answer: no it’s undecidable
  – The famous “halting problem”
    • only in 1938 did Turing get his Ph.D.

• **1940**: Church introduces the *simple theory of types*
Fast Forward…

• 1958 (Haskell Curry) and 1969 (William Howard) observe a remarkable correspondence:
  
<table>
<thead>
<tr>
<th>types</th>
<th>~</th>
<th>propositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>programs</td>
<td>~</td>
<td>proofs</td>
</tr>
<tr>
<td>computation</td>
<td>~</td>
<td>simplification</td>
</tr>
</tbody>
</table>

• 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 $F_\omega$
• 1972: Per Martin-Löf introduces intuitionistic type theory
• 1974: John Reynolds independently discovers System F

Basis for modern type systems: OCaml, Haskell, Scala, Java, C#, …
… 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 $F\omega$
- **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 Caml

- **1990’s**: up to Coq version 6.2
- **2000-2010**: Coq version 8.3
- **2011**: Coq version 8.4 ← CIS 500

Too many contributors to mention here…

CIS 500: Software Foundations
Fall 2013
So much for foundations… what about software?
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 to disastrous consequences?
Approaches to Reliability

• Social
  – Code reviews
  – Extreme/Pair programming

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

• Technological
  – “lint” tools
  – Fuzzers

• Mathematical
  – Sound type systems
  – “Formal” verification

Less “formal”: Techniques may miss problems in programs

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

Even the most “formal” can still have holes:
• did you prove the right thing?
• do your assumptions match reality?

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

1. basic tools from **logic** for making and justifying precise claims about programs

2. the use of **proof assistants** to construct rigorous, machine checkable, logical arguments

3. the idea of **functional programming**, both as a method of programming and as a bridge between programming and logic

4. techniques for formal **verification** of properties of specific programs

5. the use of **type systems** for establishing well-behavedness guarantees for all programs in a given language
Can it Scale?

- Use of theorem proving to verify “real” software is still considered to be the bleeding edge of PL research.

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

- **Ynot** – verified DBMS, web services
  Morrisett, Harvard

- **Verified Software Toolchain**
  Appel, Princeton

- **Bedrock**
  Chlipala, MIT

- **CertiKOS** – certified OS kernel
  Shao & Ford, Yale

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

Type System and SSA
Operational Semantics
Syntax
Memory Model
Proof Techniques & Metatheory

Coq

Extract

OCaml Bindings
Parser
Printer

LLVM

C Source Code
LLVM IR
Transform
LLVM IR
Other Optimizations
Target
Does it work?

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

Random test-case generation

Source Programs

GCC

79 bugs: 25 critical

LLVM

202 bugs

Open

{8 other C compilers}

325 bugs in total

Verified Compiler: CompCert [Leroy et al.]

<10 bugs found in unverified front-end component
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.*

(emphasis mine)
Why CIS 500?

- **Foundations**
  - Functional programming
  - Constructive logic
  - Logical foundations
  - Proof techniques for inductive definitions

- **Semantics**
  - Operational semantics
  - Modeling imperative “While” programs
  - Hoare logic for reasoning about program correctness

- **Type Systems**
  - Simply typed λ-calculus
  - Type safety
  - Subtyping
  - Dependently-typed programming

- **Coq interactive theorem prover**
  - turns doing proofs & logic into programming fun!
Coq in CIS 500

- We’ll use Coq version 8.4
  - Available on CETS systems
  - Easy to install on your own machine

- See the web pages at: coq.inria.fr

- Two different user interfaces
  - CoqIDE – a standalone GUI / editor
  - ProofGeneral – an Emacs-based editing environment

- Course web pages have more information.
Coq’s Full System
Subset Used in CIS 500

To start.

By the end of the semester.
Getting acquainted with Coq.
CIS 500: TODO

• Soon:
  – Register for Piazza
  – Try to log in to Canvas
  – Reading: Preface and Basics

• Before next time:
  – Install Coq v. 8.4
  – Buy a clicker from the bookstore

• HW1: Basics.v
  – Due: Thursday, Sept. 5th at 8:00pm
    • This is Rosh Hashanah – if that’s a problem, come talk to me.
  – Available on the web pages
  – Complete all non-optional exercises
  – There are no “advanced” for this HW
  – Submit to Canvas