CIS 500
Software Foundations

Fall 2007

Lecture 1

Course Overview

What is "software foundations"?

Software foundations (or "theory of programming languages") is the mathematical study of the **meaning** of programs.

The goal is finding ways to describe program behaviors that are both **precise** and **abstract**.

- **precise** so that we can use mathematical tools to formalize and check interesting properties
- **abstract** so that properties of interest can be discussed clearly, without getting bogged down in low-level details

Why study software foundations?

- To prove specific properties of particular programs (i.e., program verification)
  - Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive
- To develop intuitions for *informal* reasoning about programs
- To prove general facts about all the programs in a given programming language (e.g., safety or isolation properties)
- To deeply understand language features (and their interactions) and develop principles for better language design *(PL is the "materials science" of computer science...)*

Approaches to Program Meaning

- *Denotational semantics* and *domain theory* view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
- *Program logics* such as *Hoare logic* and *dependent type theories* focus on logical rules for reasoning about programs.
- *Operational semantics* describes program behaviors by means of *abstract machines*. This approach is somewhat lower-level than the others, but is extremely flexible.
- *Process calculi* focus on the communication and synchronization behaviors of complex concurrent systems.
- *Type systems* describe *approximations* of program behaviors, concentrating on the shapes of the values passed between different parts of the program.

Overview

We will concentrate on operational techniques and type systems.

1. Part I: Foundations
   1.1 Functional programming
   1.2 Inductive definitions and proof techniques
   1.3 Constructive logic **new**
   1.4 The Coq proof assistant **new**
2. Basics
   2.1 Operational semantics
   2.2 The lambda-calculus
3. Type systems
   3.1 Simply typed lambda-calculus
   3.2 Type safety
   3.3 Subtyping
4. Case study
   4.1 Featherweight Java

What you can expect to get out of the course

- A more sophisticated perspective on programs, programming languages, and the activity of programming
  - How to view programs and whole languages as formal, mathematical objects
  - How to make and prove rigorous claims about them
  - Detailed study of a range of basic language features
- Powerful mathematical tools for language (and software) design, description, and analysis
  - Constructive logic
  - Inductive methods of definition and proof
- Expertise using a cutting-edge mechanical proof assistant

Most software designers are language designers, at some point!

What this course is not

- An introduction to programming (see CIT 591)
- A course on functional programming (though we'll be doing some functional programming along the way)
- A course on compilers (you should already have basic concepts such as lexical analysis, parsing, abstract syntax, and scope under your belt)
- A comparative survey of many different programming languages and styles (boring!)

Administrative Stuff

Personnel

Instructor:	Benjamin Pierce Levine 562 `bcpierce@cis.upenn.edu` Office hours: Wed, 3:00–5:00 (subject to change!)
Teaching Assistant:	Leonid Spesivtsev Levine 475 Office hours: TBA
Administrative Assistant:	Kamila Dyjas Mauro Levine 311

Information

Textbook:	Types and Programming Languages, Benjamin C. Pierce, MIT Press, 2002
Webpage:	`http://www.seas.upenn.edu/~cis500`
Mailing list:	**To be announced**

Exams

1. **First mid-term:** Wednesday, October 10, in class
2. **Second mid-term:** Wednesday, November 14, in class
3. **Final:** Wednesday, December 18, 12-2 PM

Grading

Final course grades will be computed as follows:

- Homework: 20%
- 2 midterms: 20% each
- Final: 40%

(Lack of) extra Credit

1. Grade improvements can *only* be obtained by sitting in on the course next year and turning in all homeworks and exams. (If you are doing this to improve your grade from last year, please speak to me after class so I know who you are.)
2. There will be no extra credit projects, either during the semester or after the course ends. Concentrate your efforts on this course, now.

Collaboration

- Collaboration on homework is *strongly encouraged*
- Studying with other people is the best way to internalize the material
- Form study groups!
  - 2 people is the ideal size.
  - 3 is less good.
  - 4 is too many.

"You never really misunderstand something until you try to teach it..."
— Anon.

Homework

- Small part of your grade, but a large part of your understanding — impossible to perform well on exams without seriously grappling with the homework
- Submit one assignment per study group
- On written parts of homeworks, we will grade a semi-random subset of the problems on each assignment
- Some solutions are in the back of the book. Write your answer down *before* looking
- Late (non-)policy: Homework will *not be accepted* after the announced deadline

First Homework Assignment

- The first homework assignment (on basic functional programming) is **due next Monday by 10AM**.
- You will need:
  - An account on a machine where Coq is installed (you can also install Coq on your own machine if you like)
  - Instructions on running Coq
    - To be presented in recitations on Friday
  - The Homework Instructions handout
  - The lecture script from today (available from the Schedule page in the course web pages)

Recitations

- Recitations will start this week; they will take place on Fridays
- Purpose of recitations:
  - Hands-on help with Coq proof assistant
  - Deepening concepts from main lectures
  - Discussion of additional examples, etc.

The WPE-I

- PhD students in CIS must pass a five-section Written Preliminary Exam (WPE-I)
  Software Foundations is one of the five areas
- The final for this course is also the software foundations WPE-I exam
- Near the end of the semester, you will be given an opportunity to declare your intention to take the final exam for WPE credit

The WPE-I (continued)

- You do not need to be enrolled in the course to take the exam for WPE credit
- If you are enrolled in the course and also take the exam for WPE credit, you will receive two grades: a letter grade for the course final and a Pass/Fail for the WPE
- You can take the exam for WPE credit even if you are not currently enrolled in the PhD program

The Coq Proof Assistant

What is a Proof?

- A proof is an indisputable argument for the truth of some mathematical assertion
- Proofs are ubiquitous in most branches of computer science
- However, unlike proofs in pure mathematics, many CS proofs are very long, shallow, and boring
- Can computers help?

What is a Proof Assistant?

Different ways of proving theorems with a computer:

- **Automatic theorem provers** find complete proofs on their own
  - Huge amount of work, beginning in the AI community in the 50s to 80s
  - In limited domains, extremely successful (e.g., hardware model checking)
  - In general, though, this approach is just too hard
- **Proof checkers** simply *verify* proofs that they are given
  - These proofs must be presented in an extremely detailed, low-level form
- **Proof assistants** are hybrid tools
  - "Hard steps" of proofs (the ones requiring deep insight) are provided by a human
  - "Easy parts" are filled in automatically

Some Proof Assistants

There are now a number of mature, sophisticated, and widely used proof assistants...

- Mizar (Poland)
- PVS (SRI)
- ACL2 (U. Texas)
- Isabelle (Cambridge / Munich)
- Coq (INRIA)
- ...

The Coq Proof Assistant

- Developed at the INRIA research lab near Paris over the past 20 years
- Based on an extremely expressive logical foundation
  - The *Calculus of Inductive Constructions*
  - (hence the name!)
- Has been used to check a wide range of significant mathematical results
  - E.g., Gonthier's fully verified proof of the four-color theorem

Why Use Coq in This Course

- Rigor
  - using Coq forces us to be completely precise about the things we define and the claims we make about them
  - Coq's core notions of inductive definition and proof are a good match for the fundamental ways we define and reason about programming languages
- Interactivity
  - Instant feedback on homework
  - Easy to experiment with consequences of changing definitions, different reasoning techniques, etc.
- Useful background
  - Proof assistants are being used more and more widely in industry and academia
- Fun!
  - Coq is pretty addictive...

Functional Programming in Coq

Functional Programming

- Many interesting mathematical concepts can be expressed as *computations*.
- Coq includes a small programming language that can be used for this purpose
- This is a *functional* programming language — a baby cousin of well-known functional languages such as Scheme, Haskell, OCaml, and Standard ML.

Functional Programming

The **functional style** can be described as a combination of...

- *persistent* data structures (which, once built, are never changed)
- *recursion* as a primary control structure
- heavy use of *higher-order functions* (functions that take functions as arguments and/or return functions as results)

Imperative languages, by contrast, emphasize...

- *mutable* data structures
- *looping* rather than recursion
- *first-order* rather than higher-order programming (though many object-oriented design patterns involve higher-order idioms—e.g., Subscribe/Notify, Visitor, etc.)