CIS 500

Software Foundations

Fall 2002

2 December

Administrivia

- Final HW assignment (#13) is now available; due next Monday
- Recitations
  - Recitations as usual this week
  - No recitations after this week (extra office hours instead)
- Class
  - Today: recap and discussion
  - Wednesday: one more object model (optional lecture)
  - Next Monday: review session (come with questions)
- Exam: Dec 20, 11-1
  - Location: Steinberg-Dietrich Hall (3620 Locust Walk), Room 350
  - Coverage: Chapters 1 to 19 of TAPL, excluding 12 and 15.6, plus reading knowledge of basic OCaml

Recap... Where we've been

What is "software foundations"?

Software foundations (a.k.a. "theory of programming languages") is the study of the meaning of programs.

A main goal is finding ways to describe program behaviors that are both precise and abstract.

Why study software foundations?

- To be able to prove specific facts about particular programs (i.e., program verification)

  Important in some domains (safety-critical systems, hardware design, inner loops of key algorithms, ...), but (inherently?) difficult and expensive. We have not said much about this in the course.
- 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 security properties)
- To understand language features (and their interactions) deeply and develop principles for better language design

  PL as the "materials science" of computer science...

What I hope you got 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
- Deep intuitions about key language properties such as type safety
- Powerful tools for language design, description, and analysis

N.b.: most software designers are language designers!

Overview

In this course, we concentrated on operational semantics and type systems.

- Part O: Background
  - A taste of OCaml
  - Functional programming style
- Part I: Basics
  - Inductive definitions and proofs
  - Operational semantics
  - The lambda-calculus
  - Evaluator implementation in OCaml

- Part II: Type systems
  - Simple types
  - Type safety — preservation and progress
  - Formal description of a variety of basic language features (records, variants, lists, casting, ...)
  - References
  - Exceptions
  - Subtyping
  - Metatheory of subtyping (subtyping and typechecking algorithms)
- Part III: Object-oriented features (case studies)
  - A simple imperative object model
  - An direct formalization of core Java

What next?

The Research Literature

With this course under your belt, you should be ready to directly address research papers in programming languages.

This is a big area, and each sub-area has its own special techniques and notations, but you now have the basic intuitions needed to grapple with these on your own.

The rest of TAPL

Several more "core topics" are covered in the second half of TAPL.

- Recursive types (including a rigorous treatment of induction and co-induction)
- Parametric polymorphism (universal and existential types)
  - Bounded quantification
  - Refinement of the imperative object model
  - ML-style type inference
- Type operators
  - Higher-order bounded quantification
  - A purely functional object model