17-355/17-665: Program Analysis

Spring 2017 (we hope to offer the course in future Spring semesters, if demand permits)

Instructor: Jonathan Aldrich

Description

This course covers both foundations and practical aspects of the automated analysis of programs, which is becoming increasingly critical to find software errors and assure program correctness. The theory of abstract interpretation captures the essence of a broad range of program analyses and supports reasoning about their correctness. Building on this foundation, the course will describe program representations, data flow analysis, alias analysis, interprocedural analysis, dynamic analysis, and symbolic execution. Through assignments and projects, students will design and implement practical analysis tools that find bugs and verify properties of software.

Prerequisites: 15-251 and (15-150 or 15-214)

Rationale: students must have some mathematical maturity in order to understand and use the theory of abstract interpretation to reason about the analyses they define; this justifies requiring 15-251. Furthermore, students must have some programming maturity in order to implement program analyses. In particular, familiarity with higher-order programming frameworks is important to developing analysis implementations on top of software (such as the Soot analysis framework) used in the course. Both 15-150 and 15-214 cover higher-order programming, though in different contexts, so either can be used to satisfy the second prerequisite.

Units: 12 units.

Note: 12 units is likely more appropriate than 9 as the course will include substantial programming projects (including a somewhat open-ended final project) in which students will implement program analyses.
Learning goals

- Students will learn to use abstract interpretation to formally prove that an analysis is sound and will terminate. They will also learn to analyze the scalability of a program analysis and compare the precision of different analysis techniques.

- Students will learn to implement program analyses that verify program properties and find bugs using dataflow analysis, interprocedural analysis, alias analysis, and symbolic execution. Furthermore, they will learn how program analysis frameworks can leverage abstract interpretation ideas in order to reuse the vast majority of analysis boilerplate.

Topic overview

Introduction to program analysis
Program representation
Dataflow analysis and abstract interpretation
  - Basic dataflow analysis
  - Abstract interpretation frameworks
  - The worklist algorithm
  - Widening
Interprocedural analysis
  - Approaches to context sensitivity
  - Call-graph construction
Analysis of functional programs [note: consider interaction with prerequisites]
  - Reformulation of abstract interpretation
  - Control-flow analysis
Analyzing heap-manipulating programs
  - Approaches to abstracting heap locations: pointer analysis, alias analysis, shape analysis
Symbolic execution
  - Connection to abstract interpretation
Dynamic analysis
  - Concolic testing (combination of dynamic analysis and symbolic execution)
Design tradeoffs in program analyses
  - Effect of different languages
  - Heuristic vs. sound analysis
Applications of program analysis (may be spread out somewhat in the above lectures)
  - Concurrency, e.g. race condition or deadlock analysis
  - Security, e.g. information flow analysis
  - Correct API use, e.g. typestate analysis
Assignments (tentative)

1. Specify a simple program analysis (TBD, motivated by verification or bug-finding). Prove that the analysis terminates. Implement the analysis, including the worklist algorithm.

2. Specify a second program analysis using abstract interpretation theory. Formally show both termination and correctness. Implement the analysis leveraging an abstract interpretation-based framework.

3. Extend one of the analyses above to work interprocedurally, specifying and implementing an appropriate context sensitivity approach.

4. Extend one of the analyses above to reason about flows through the heap.

5. Implement a simple form of symbolic execution

6. Final project: with a partner, design and implement your own program analysis. The instructor will provide several suggestions but ambitious students may propose their own ideas.

Prior Offerings and Materials

Although this course has not been offered at CMU at the undergraduate level, a similar course, 15-819 O was offered to graduate students (mostly at the Ph.D. level) in Spring 2010 (Aldrich), Spring 2013 (Aldrich), and Spring 2016 (Le Goues). The Spring 2013 course web page includes homework assignments and an extensive set of course notes that will be adapted for the undergraduate offering of the course:

http://www.cs.cmu.edu/~aldrich/courses/15-819O-13sp/