Design and Analysis of Programming Languages

ECS 240

Administrivia

- Who am I?
- Website: http://www.cs.ucdavis.edu/~su/teaching/ecs240-w17
  - SmartSite & Piazza
  - Will post there announcements, lectures, assignments, etc.
- Office hours: Th 1-2 PM, 3011 Kemper (reserved for ECS 240)
  - Also Tu/Th 2-3 PM *if I'm not helping ECS 140A students*
- TA & TA office hours: Nima Johari
  - Tu 11-12:30 PM (3016 Kemper)
  - W 11-noon (3106 Kemper)
  - F 1-2:30 PM (53 Kemper)

Course Work

- Lectures
- Homework
  - Concentrated in the first half of the course (3-4)
  - Mostly theoretical in nature (tool introduction)
- Project
  - Concentrated in the second half of the course
  - I will suggest some topics and you are free to propose your own
  - You select a topic (best: connect with your current research)
  - Project report and presentation (dates TBD)
  - Take-home final (date TBD)
- Grading (tentative):
  - Class participation (~10%)
  - Homework (~30%)
  - Take-home final (~20%)
  - Project (~40%)

Prerequisites

- Programming experience
  - exposure to various language constructs and their meaning
  - e.g., C, Java, C++, ML, Lisp, Prolog
  - e.g., ECS 140A, 142 or equivalent

- Mathematical maturity
  - we'll use formal notation to describe the meaning of programs
  - e.g., set theory, formal proofs, induction
    - Chapter 1 in Winskel's book

- If you don't have either, are an undergraduate, or are from another department, please see me

Contemporary Landscape

- Programming languages is one of the oldest CS fields
- And one of the most vibrant today!
- Current trends
  - Type safety gaining acceptance as a viable security component
  - Modern program analysis becoming a major component of software engineering
  - Renewed interest in language design and parallelism
  - Programming synthesis for education and end-user programming

Course Goals

- Learn techniques for language/program analysis
  - formal semantics (operational, axiomatic, ~~denotational~~)
  - reasoning about program behavior
  - case studies of languages and features

- Discuss practical applications of these techniques
  - software engineering
  - security

Course Readings

- Mostly classical and recent research papers
- Other references:
  - Glynn Winskel, “The Formal Semantics of Programming Languages”
  - John Mitchell, “Foundations for Programming Languages”
  - Benjamin Pierce, “Types and Programming Languages”

Topic I: Language Specification

- Three pedigreed approaches:
  - Operational semantics (how?)
    - rules for execution on an abstract machine
    - useful for implementing a compiler or interpreter
  - Axiomatic semantics (why?)
    - logical rules for reasoning about the behavior of a program
    - useful for proving program correctness
  - Denotational semantics (what?) [*will skip this time*]
    - meaning described as a function from programs to elements of a domain

- Why isn't semantics used on a mass scale?

Why Don't People Use Semantics?

- Semantics is fairly heavyweight and not (yet) cost-effective
  - For everyday (and everyone's) use.
  - Notation is sometimes dense
- Semantics is general and explains:
  - For all possible inputs x, the output is y and the state changes so that ...
- Most programmers are content to know:
  - What is the output for the particular input I will test this program on?

- But who then definitely needs semantics?

Who Needs Semantics

- Those who want to describe unambiguously a language feature or a program transformation:
  - Semantics is the basis for most formal arguments in PL research
  - Semantics is a standard tool in PL research
- Those who write programs that must work for all inputs:
  - program transformation and instrumentation tools
  - program analyzers
  - software engineering tools
  - compilers and interpreters
  - critical software

Topic II: Language Design

- Languages are adopted to fill a void
  - Enable a previously difficult/impossible application
  - Orthogonal to language design quality (almost)

- Programmer training is the dominant adoption cost
  - Languages with many users are replaced rarely
  - Popular languages become ossified
  - But easy to start in a new niche . . .

Why So Many Languages?

- Many languages were created for specific applications
- Application domains have distinctive (and conflicting) needs
  - leading to a proliferation of languages
- Examples:
  - Artificial intelligence: symbolic computation (Lisp, Prolog)
  - Scientific Computing: high performance (Fortran)
  - Business: report generation (COBOL)
  - Systems programming: low-level access (C)
  - Customization: scripting (Perl, ML, Javascript, TCL)
  - Distributed systems: mobile computation (Java)
  - Special purpose languages: …

Language Paradigms

- Imperative
  - Fortran, Algol, Cobol, C, Pascal
- Functional
  - Lisp, Scheme, ML, Haskell
- Object oriented
  - Smalltalk, Eiffel, Self, C++, Java, Javascript
- Logic
  - Prolog, λProlog, Datalog
- Concurrent
  - Erlang, X10, Fortress
- Special purpose
  - TEX, SQL, PostScript, HTML

What Makes a Good Language?

- No universally accepted metrics for design
- "A good language is one people use" ?
- NO !
  - Is COBOL the best language?

Good Language Features

- Simplicity (syntax and semantics)
- Readability
- Safety
- Support for programming in the large
- Efficiency (of execution and compilation)
- Support for abstraction (high level)

Good Languages

- These goals almost always conflict
- Examples:
  - Safety checks cost something in either compilation or execution time
  - Safety and machine independence may exclude efficient low-level operations
  - Type systems restrict programming style in exchange for strong guarantees

Story: The Clash of Two Features

- Real story about bad programming language design
- Cast includes famous scientists
- ML ('82) functional language with polymorphism and monomorphic references (i.e., pointers)
- Standard ML ('85) innovates by adding polymorphic references
- It took 10 years to fix the "innovation"

Polymorphism (Informal)

- Code that works uniformly on various types of data
- Examples:

  length : α list → int

  hd : α list → α

  snd : $\alpha \times \beta \rightarrow \beta$

- Type inference:
  - generalize all elements of the input type that are not used by the computation
  - instantiation: if $e : \tau$ then $e : [\tau' / \alpha]\tau$ (substitute τ' for α in τ)

References in Standard ML

- Like “updatable pointers” in C
- Type constructor: τ * (this is not the real ML notation)
- Expressions:

  new : $\tau \to \tau$ * (allocate a cell to store a τ)
  *e : τ when e : τ * (read through a pointer)
  *e := e’ with e : τ * and e’ : τ (write through a pointer)

- Works just as you might expect

Polymorphic References: A Major Pain

Consider the following program fragment:

Code	Type inference
fun id(x) = x	id : $\alpha \rightarrow \alpha$ (for any α)
val c = new id	c : $(\alpha \rightarrow \alpha)$ * (for any α)
fun inc(x) = x + 1	inc : int → int
*c := inc	Ok, since c : (int → int) *
(*c) (true)	Ok, since c : (bool → bool) *

Reconciling Polymorphism and References

- The type system fails to prevent a type error!

- Solutions:
  - e.g., weak type variables:
    - polymorphic variables whose instantiation is restricted
    - difficult to use, several failed proofs of soundness
  - value restriction: generalize only the type of <u>values</u>!
    - easy to use, simple proof of soundness

Story: Java Bytecode Subroutines

- Java bytecode programs contain subroutines (jsr) that run in the caller's stack frame
- jsr complicates the formal semantics of bytecode
  - Several verifier bugs were in code implementing jsr
  - 30% of typing rules, 50% of soundness proof due to jsr
- It is not worth it
  - In 650K lines of Java code, 230 subroutines, saving 2427 bytes, or 0.02%
  - 13 times more space could be saved by renaming the language to Oak

Language Design Lessons

- Good language design is hard
  - Rarely, if ever, achieved by accident
- Simplicity is rare in practice
- Real languages are isolated points in a huge design space
- PL research considers tiny languages (e.g., λ-calculus) to separate and study core issues in isolation
- In practice, we must also pay attention to the language as a whole

Topic III: Applications of Semantic Tools

- You might not end up doing research in semantics but it is very likely that you will need to apply some of the techniques in your research
- We may discuss a few sample applications, e.g.
  - Software model checking
  - Vulnerability detection
  - Verifying dimensional unit correctness

Next time

- IMP & operational semantics