

## Design and Analysis of Programming Languages

CS263 - Spring 07

CS 263 Lecture 1

1

### Course Work

---

- Lectures
- Homeworks
  - Six or seven, concentrated in the first half of the course
  - Mostly theoretical in nature
- Projects
  - Concentrated in the second half of the course
  - I will suggest some topics and you are free to propose others
  - You select the topic (surveys, programming, research)
  - Each team will write a report and give a 15 minute talk
- Final exam
  - Take home, before the end of the semester

CS 263 Lecture 1

3

### Contemporary Landscape

---

- Programming languages is one of the oldest CS fields.
- And one of the most vibrant today!
- Current trends:
  - Web/Java renews interest in language design
  - A lot of need for domain specific languages
  - Type safety gains acceptance as a viable security component
  - Modern program analysis will become a major component of software engineering in 5 years

CS 263 Lecture 1

5

### Administrivia

---

- Who am I?
- Web page
  - <http://www.cs.berkeley.edu/~necula/cs263>
  - find there announcements, lectures, notes, assignments, exams, solutions, etc.
- Office hours
  - Monday 1-2pm, and by appointment
- Invited speakers
  - Sometimes jointly with the PS seminar (Monday 4-5:30pm)
  - Today: "Resource-Aware Programming" - 4pm, 540 Cory
    - Walid Taha, Rice Univ.

CS 263 Lecture 1

2

### Prerequisites

---

- Programming experience
  - exposure to various language constructs and their meaning
  - e.g., C, Java, C++, ML, Lisp, Prolog
  - e.g., CS164 (undergrad compiler class) or equivalent
- Mathematical maturity
  - we'll use formal notation to describe the meaning of programs
  - e.g., set theory, formal arguments, induction
    - Chapter 1 in Winskel's book
  - hardest topic: Denotational semantics, Chapter 5 and 8
- If you don't have either, are an undergraduate, or from another department, please see me.

CS 263 Lecture 1

4

### Course Goals

---

- Learn techniques for language/program analysis
  - formal semantics (operational, axiomatic, denotational)
  - reasoning about program behavior and program analyses
  - case studies of languages and features
- Discuss practical applications of these techniques
  - security, program analysis, program verification
- Introduction to current research on programming languages
- Cover part of the syllabus for the PL prelim. exam

CS 263 Lecture 1

6

## Course Readings

---

- **Part I: Semantics, evaluation strategies, imperative languages:**
  - Glynn Winskel, "The Formal Semantics of Programming Languages"
- **Part II: Type theory, typed  $\lambda$ -calculus, functional languages**
  - Benjamin Pierce, "Types and Programming Languages"
  - Also by Pierce: "Advanced Topics in Types and PL"
  - Good reference books for PL researchers
- **Part III: Applications**
  - Mostly research papers

CS 263 Lecture 1

7

## Topic I: Language Specification

---

- **Three pedigreed approaches:**
  - **Operational semantics**
    - rules for execution on an abstract machine
    - useful for implementing a compiler or interpreter
  - **Axiomatic semantics**
    - logical rules for reasoning about the behavior of a program
    - useful for proving program correctness
  - **Denotational semantics**
    - meaning described as a function from programs to elements of a domain
- **Why isn't semantics used on a mass scale?**

CS 263 Lecture 1

8

## Why Don't They Use Semantics?

---

- **Semantics is fairly heavyweight and not (yet) cost-effective**
  - For everyday (and everyone's) use.
  - Notation is sometimes dense
- **Semantics is general. It 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 inputs I will test this program on?
- **But who then definitely needs semantics?**

CS 263 Lecture 1

9

## Who Needs Semantics

---

- **Those who want to describe unambiguously a language feature or a program analysis/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

CS 263 Lecture 1

10

## 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 . . .

CS 263 Lecture 1

11

## What Makes a Good Language?

---

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

CS 263 Lecture 1

12

## Good Language Features

---

- Simplicity (syntax and semantics)
- Readability
- Safety
- Support for programming large systems
- Efficiency (of execution and compilation)

CS 263 Lecture 1

13

## Good Languages

---

- These goals almost always conflict.
- Examples:
  - Safety checks cost something in either compilation or execution time.
  - Type systems restrict programming style in exchange for strong guarantees.

CS 263 Lecture 1

14

## 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"

CS 263 Lecture 1

15

## Polymorphism (Informal)

---

- Code that works uniformly on various types of data
- Examples:
  - `length : list  $\alpha$   $\rightarrow$  int` (takes an argument of type list of  $\alpha$ , returns an integer, for any type  $\alpha$ )
  - `head : list  $\alpha$   $\rightarrow$   $\alpha$`
- 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  $\tau$  for  $\alpha$  in  $\tau$ )

CS 263 Lecture 1

16

## References in Standard ML

---

- Like "pointers" in C
- Type constructor: `ptr  $\tau$`
- Expressions:
  - `alloc :  $\tau$   $\rightarrow$  ptr  $\tau$`  (allocate a cell to store a  $\tau$ )
  - `*e :  $\tau$  when e : ptr  $\tau$`  (read through a pointer)
  - `*e := e' when e : ptr  $\tau$  and e' :  $\tau$`  (write through a pointer)
- Works just as you might expect

CS 263 Lecture 1

17

## Polymorphic References: A Major Pain

---

Consider the following program fragment:

Code	Type inference
<code>fun id(x) = x</code>	<code>id : <math>\alpha</math> <math>\rightarrow</math> <math>\alpha</math></code> (for any $\alpha$ )
<code>val c = alloc id</code>	<code>c : ptr (<math>\alpha</math> <math>\rightarrow</math> <math>\alpha</math>)</code> (for any $\alpha$ )
<code>fun inc(x) = x + 1</code>	<code>inc : int <math>\rightarrow</math> int</code>
<code>*c := inc</code>	Ok, since <code>c : ptr (int <math>\rightarrow</math> int)</code>
<code>(*c) (true)</code>	Ok, since <code>c : ptr (bool <math>\rightarrow</math> bool)</code>

CS 263 Lecture 1

18

## Reconciling Polymorphism and References

---

- The type system fails to prevent a type error!
- Solutions:
  - weak type variables:
    - polymorphic variables whose instantiation is restricted
    - difficult to use, several failed proofs of soundness
  - value restriction: generalize only the type of values!
    - easy to use, simple proof of soundness

CS 263 Lecture 1

19

## Story: Java Bytecode Subroutines

---

- Java bytecode programs contain subroutines (jsr) that run in the caller's stack frame
- jsr complicates the formal semantics of bytecodes
  - 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

CS 263 Lecture 1

20

## 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.,  $\lambda$ -calculus) to separate and study core issues in isolation
- In practice, we must also pay attention at the language as a whole

CS 263 Lecture 1

21