CS 4110

Programming Languages & Logics

Lecture 1
Course Overview

JavaScript

```
[] + []
{} + []
[] + {}
{} + {}
```

From *Wat:*
`https://www.destroyallsoftware.com/talks/wat`

Java

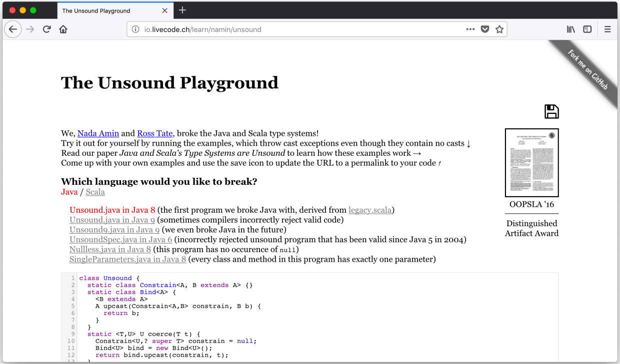
```
class A {
    static int a = B.b + 1;
}

class B {
    static int b = A.a + 1;
}
```

Python

```
a = [1], 2
a[0] += 3
```

Java and Scala



Nada Amin and Ross Tate:

`http://io.livecode.ch/learn/namin/unsound`

Design Desiderata

Question: What makes a good programming language?

Design Desiderata

Question: What makes a good programming language?

One answer: A good language is one people use.

Design Desiderata

Question: What makes a good programming language?

One answer: A good language is one people use.

Wrong! Is JavaScript bad? What's the best language?

Design Desiderata

Question: What makes a good programming language?

One answer: A good language is one people use.

Wrong! Is JavaScript bad? What's the best language?

Some good features:

- Simplicity (clean, orthogonal constructs)
- Readability (elegant syntax)
- Safety (guarantees that programs won't "go wrong")
- Modularity (support for collaboration)
- Efficiency (it's possible to write a good compiler)

Design Challenges

Unfortunately these goals almost always conflict.

- Types provide strong guarantees but restrict expressiveness.
- Safety checks eliminate errors but have a cost—either at compile time or run time.
- A language that's good for quick prototyping might not be the best for long-term development.

Design Challenges

Unfortunately these goals almost always conflict.

- Types provide strong guarantees but restrict expressiveness.
- Safety checks eliminate errors but have a cost—either at compile time or run time.
- A language that's good for quick prototyping might not be the best for long-term development.

A lot of research in programming languages is about discovering ways to gain without (too much) pain.

Language Specification

Formal Semantics: what do programs mean?

Three Approaches

- Operational
  - Models program by its execution on abstract machine
  - Useful for implementing compilers and interpreters
- Axiomatic
  - Models program by the logical formulas it obeys
  - Useful for proving program correctness
- Denotational
  - Models program literally as mathematical objects
  - Useful for theoretical foundations

Language Specification

Formal Semantics: what do programs mean?

Three Approaches

- Operational
  - Models program by its execution on abstract machine
  - Useful for implementing compilers and interpreters
- Axiomatic
  - Models program by the logical formulas it obeys
  - Useful for proving program correctness
- Denotational
  - Models program literally as mathematical objects
  - Useful for theoretical foundations

Question: few languages have a formal semantics. Why?

Formal Semantics

Too Hard?

- Modeling a real-world language is hard
- Notation can gets very dense
- Sometimes requires developing new mathematics
- Not yet cost-effective for everyday use

Overly General?

- Explains the behavior of a program on *every* input
- Most programmers are content knowing the behavior of their program on *this* input (or these inputs)

Okay, so who needs semantics?

Who Needs Semantics?

Unambiguous Description

- Anyone who wants to design a new feature
- Basis for most formal arguments
- Standard tool in PL research

Exhaustive Reasoning

- Sometimes have to know behavior on all inputs
- Compilers and interpreters
- Static analysis tools
- Program transformation tools
- Critical software

Story: Unexpected Interactions

A real story illustrating the perils of language design

Cast of characters includes famous computer scientists

Timeline:

- 1982: ML is a functional language with type inference, polymorphism (generics), and monomorphic references (pointers)
- 1985: Standard ML innovates by adding polymorphic references $\rightarrow$ unsoundness
- 1995: The “innovation” fixed

ML Type System

Polymorphism: allows code to be used at different types

Examples:

- List.length : $\forall \alpha.\ \alpha$ list $\rightarrow$ int
- List.hd : $\forall \alpha.\ \alpha$ list $\rightarrow \alpha$

Type Inference: $e \rightsquigarrow \tau$

- e.g., let $id\ (x) = x \rightsquigarrow \forall \alpha.\ \alpha \rightarrow \alpha$
- Generalize types not constrainted by the program
- Instantiate types at use id (true) $\rightsquigarrow$ bool

ML References

By default, values in ML are immutable.

But we can easily extend the language with imperative features.

Add reference types of the form τ ref

Add expressions of the form

ref $e : \tau$ ref	where $e : \tau$	(allocate)
$!e : \tau$	where $e : \tau$ ref	(dereference)
$e_1 := e_2$: unit	where $e_1 : \tau$ *ref* and $e_2 : \tau$	(assign)

Works as you'd expect (like pointers in C).

Polymorphism + References

Consider the following program

Code	Type Analysis
let id = (fun x -> x)	

Polymorphism + References

Consider the following program

Code	Type Analysis
let id = (fun x -> x)	
let p = ref id	

Polymorphism + References

Consider the following program

Code	Type Analysis
let id = (fun x -> x)	
let p = ref id	
let inc = (fun n -> n+1)	

Polymorphism + References

Consider the following program

Code	Type Analysis
let id = (fun x -> x)	
let p = ref id	
let inc = (fun n -> n+1)	
p := inc;	
(!p) true	

Polymorphism + References

Consider the following program

Code	Type Analysis
let id = (fun x -> x)	id : $\alpha \to \alpha$
let p = ref id	
let inc = (fun n -> n+1)	
p := inc;	
(!p) true	

Polymorphism + References

Consider the following program

Code	Type Analysis
let id = (fun x -> x)	id : $\alpha \to \alpha$
let p = ref id	p : $(\alpha \to \alpha)$ ref
let inc = (fun n -> n+1)	
p := inc;	
(!p) true	

Polymorphism + References

Consider the following program

Code	Type Analysis
let id = (fun x -> x)	id : $\alpha \to \alpha$
let p = ref id	p : $(\alpha \to \alpha)$ ref
let inc = (fun n -> n+1)	inc : int $\to$ int
p := inc;	
(!p) true	

Polymorphism + References

Consider the following program

Code	Type Analysis
let id = (fun x -> x)	id : $\alpha \to \alpha$
let p = ref id	p : $(\alpha \to \alpha)$ ref
let inc = (fun n -> n+1)	inc : int $\to$ int
p := inc;	OK since p : (int $\to$ int) ref
(!p) true	OK since p : (bool $\to$ bool) ref

Polymorphism + References

Problem

- Type system is not sound
- Well-typed program $\rightarrow^*$ type error!

Polymorphism + References

Problem

- Type system is not sound
- Well-typed program $\rightarrow^*$ type error!

Proposed Solutions

1. "Weak" type variables
   - Can only be instantiated in restricted ways
   - But type exposes functional vs. imperative
   - Difficult to use

Polymorphism + References

Problem

- Type system is not sound
- Well-typed program $\rightarrow^*$ type error!

Proposed Solutions

1. “Weak” type variables
   - Can only be instantiated in restricted ways
   - But type exposes functional vs. imperative
   - Difficult to use
2. Value restriction
   - Only generalize types of values
   - Most ML programs already obey it
   - Simple proof of type soundness

Lessons Learned

- Features often interact in unexpected ways
- The design space is huge
- Good designs are sparse and don't happen by accident
- Simplicity is rare: n features $\rightarrow$ n^2 interactions
- Most PL researchers work with small languages (e.g., λ-calculus) to study core issues in isolation
- But must pay attention to whole languages too

Course Staff

Instructor

Nate Foster (he/him)

Teaching Assistants

Joshua Kaplan (he/him)
Samwise Parkinson (he/him)
Priya Srikumar (they/them)
...

Prerequisites

Mathematical Maturity

- Much of this class will involve formal reasoning
- Set theory, formal proofs, induction

Programming Experience

- Comfortable using a functional language
- For undergrads: CS 3110 or equivalent

Interest (having fun is a goal!)

If you don’t meet these prerequisites, please get in touch.

Course Website

http://www.cs.cornell.edu/courses/cs4110/2020fa/

Course Work

Homework

- 8 assignments, roughly one per week
- Can work with *one* partner
- Always due on Monday night at 11:59pm
- Automatic 48-hour extension, stiff penalties after that

Preliminary Exams (take-home)

- October 5
- November 9

Course Project

- Can work alone or with a partner
- Four phases: charter, alpha, beta, final

Participation (5% of your grade)

- Introduction survey (out now!)
- Mid-semester feedback
- Course evaluation

Academic Integrity

Some simple requests:

1. You are here as members of an academic community. Conduct yourself with integrity.

2. Problem sets must be completed with your partner, and only your partner. You must *not* consult other students, alums, friends, Google, GitHub, StackExchange, Course Hero, etc.!

3. If you aren't sure what is allowed and what isn't, please ask.

Respect in Class

We hold all communication (in class & online) to a high standard for inclusiveness. It may not target anyone for harassment, and it may not exclude specific groups.

Examples:

- Do not talk over other people.
- Do not use male pronouns when you mean to refer to people of all genders.
- Avoid language that has a good chance of seeming inappropriate to others.

If anything doesn't meet these standards, contact the instructor.

Disabilities and Wellness

- I will provide reasonable accommodations to students with documented disabilities (e.g., physical, learning, psychiatric, vision, hearing, or systemic).

- If you are experiencing undue personal or academic stress at any time during the semester (or if you notice that a fellow student is), contact me, Engineering Advising, or Gannett.