

CS 565:

Programming Languages

Spring 2009

M,W,F 1:30 -2:20

Lawson 1106

Administrivia

- Who am I?
 - Course Web Page:
 - <http://www.cs.purdue.edu/homes/suresh/565-Spring2009/>
 - Office Hours:
 - Tu, Th: 2 -- 3 PM
 - By appointment
 - Main text:
 - Types and Programming Languages,
B. Pierce, MIT Press
-

Course Work

- Lectures
 - Homeworks
 - Periodic (probably one every two weeks or so)
 - The answers to some of the questions will be available in the back of the text
 - Collaborating on homework encouraged
 - Programming Exercises
 - Will involve implementing type checkers and interpreters.
 - Code size for solutions will be small (< 250 lines), but solutions will be challenging.
 - A midterm
 - Cumulative final which will also serve as the qualifying exam
-

Prerequisites

- Programming experience/maturity
 - exposure to various language constructs
 - Java, ML, Lisp, Prolog, C
 - Undergraduate compilers and/or PL class
 - CS 352 and/or CS 456 or equivalent
 - Mathematical maturity
 - familiarity with first-order logic, set theory, graph theory, induction
 - Most important:
 - Intellectual curiosity and creativity
-

Resources

- Web page for text:
 - <http://www.cis.upenn.edu/~bcpierce/tapl>
 - Supplementary material:
 - <http://www2.imm.dtu.dk/~riis/PPA/ppasup2004.html>
 - Web page for ML implementations
 - <http://caml.inria.fr> (Caml)
 - <http://www.smlnj.org> (SML/NJ)
 - <http://www.mlton.org> (MLton)
 - Proceedings of conferences
 - POPL, PLDI, ICFP, ...
-

Background

- Our main goal is to find ways to describe the behavior of programs precisely and concisely
 - Motivation
 - Significant industry and government interest
 - Web, Java
 - Security issues
 - Complexity of modern-day applications
-

Motivation (cont)

- Prove specific facts about programs
 - Verify correctness
 - Important in mission-critical systems
 - Safety or isolation properties
 - Need an unambiguous vocabulary
 - Understand specific language features
 - Better language design
 - Guide improvements in implementations
-

Goals

- A more sophisticated appreciation of programs, their structure, and the field as a whole
 - Viewing programs as rich, formal, mathematical objects, not mere syntax
 - Define and prove rigorous claims about a program's meaning and behavior
 - Develop sound intuitions to better judge language properties
 - Develop tools to be better programmers, software designers and computer scientists
-

Non-goals

- An introduction to advanced programming techniques
 - No detailed discussion of machine implementations
 - The course will not be motivated from the perspective of a compiler writer
 - But, impact of design decisions on implementation tractability will be considered when appropriate
 - A survey of different languages
-

Topics

- Part I (Foundations): Semantic formalisms, λ -calculus, abstract interpretation, introduction to types
 - Part II (Design): Simply-typed λ -calculus, records, references, subtyping, object-based programming
 - Part III (Features): Polymorphism, abstract data types, advanced topics (e.g., concurrency, linearity, ...)
 - Part IV (Tools): Mechanized meta-theory and proof assistants
-

Semantics

- Three classical approaches:
 - Operational
 - Define programs in terms of rules that apply to a specific virtual machine
 - Useful for implementing a compiler or interpreter
 - Denotational
 - Meaning in terms of functions from syntax (program text) to domains (values)
 - Useful for describing the behavior of programs
 - Axiomatic
 - Logical rules for reasoning about programs
 - Useful for proving program correctness
-

Abstract Interpretation

- Reason about programs by “abstracting” away properties that are not of interest.
 - Type systems
 - Sign arithmetic
 - Reason about the correspondence between results produced in the abstract world with results that would be produced in the concrete world.
-

Case studies

- Lisp 1.5
 - Based on λ -calculus
 - Key aspect of the calculus is notion of substitution of free variables:
 - function $f(\text{args}) = \dots \times \dots$
 - Suppose x is not included in args . Where should the binding for x be constructed?
 - At the point where f is defined (lexical scoping)
 - At the points where f is applied (dynamic scoping)
 - Lisp 1.5 (and later dialects) chose dynamic scoping, even though it is widely agreed today that lexical scoping is the more sensible choice.
 - When do these distinctions arise? Why are the differences important?
 - Lack of formal semantics to explore the ramifications of design choice
-

Case studies

□ ML

- Interaction of types with references
 - Polymorphism: code that works uniformly on all various types of data
 - `length: α list -> int`
 - `hd: α list -> α`
 - `tl: α list -> α list`
 - Type inference
 - Assign the most general type to variables based on the contexts in which they occur
-

Case studies: ML

- References
 - Like updateable pointers in C
 - Expressions
 - $\text{ref } e: \tau \rightarrow \tau \text{ ref}$
 - $!e: \tau \text{ ref} \rightarrow \tau$
 - $e_1 := e_2: \tau \text{ ref} * \tau \rightarrow \tau \text{ ref}$

fun id(x) = x

val c = ref id

fun inc(x) = x + 1

c := inc

!c (true)

id: $\alpha \rightarrow \alpha$

c: $(\alpha \rightarrow \alpha) \text{ ref}$

inc: $\text{int} \rightarrow \text{int}$

Ok if we infer c: $(\text{int} \rightarrow \text{int}) \text{ ref}$

Ok if we infer c: $(\text{bool} \rightarrow \text{bool}) \text{ ref}$

Case studies: Java Array Types

- One of Java's design mistakes is the subtyping rule for arrays
- ▶ Given Object types T and T' , the array subtyping rule is

$$T <: T' \Rightarrow T[] <: T' []$$

if T is a subtype of T' then $T[]$ is a subtype of $T' []$

- ▶ **Example:** `String <: Object` thus
`String[] <: Object[]`
-

Case Studies: Java Array Types

- Static typing guarantees that if we have declaration

$T \ v;$

then the following holds at all times

$v \text{ instanceof } T$

- This is good because it guarantees that type errors do not occur at run-time.
- Arrays break static type safety

Case Studies: Java Array Types

- The following is a static error:

```
Thread[] appThreads = new Thread[10];
```

```
appThreads[0] = "badaboom";
```

//type error

- But the following is perfectly fine:

```
Object[] appThreads = new Thread[10];
```

```
appThreads[0] = "badaboom";
```

```
runFirstThread( (Thread[]) appThreads );
```

```
void runFirstThread( Thread[] o ) {
```

```
    o[0].run();
```

//runtime type error

Case Studies: Java Array Types

```
Object[] appThreads = new Thread[10]; //OK Thread[] <: Object[]  
appThreads[0] = "badaboom";          //OK String <: Object  
runFirstThread((Thread[]) appThreads); //appThreads instanceof  
Thread[]
```

```
void runFirstThread( Thread[] o ) {  
    o[0].run();          //OK o[0] is declared to be a  
    Thread
```

The problem is the subtyping rule

Case Studies: Java Array Types

Why have that rule ?

To write `sort(Object[])` once rather than have to rewrite it for each new class!

- but subtype polymorphism is a poor substitute for parametric polymorphism

The right solution is genericity...

`sort(<A implements Comparable<A>>)`

Lessons

- Language design is as much about safety as it is about efficiency and expressiveness.
 - Need tools and frameworks to reason about and compare different language features and designs:
 - untyped λ -calculus as a universal computation language. Precisely define its behavior using different semantic models (operational, denotational, axiomatic)
 - typed λ -calculi to express safety and abstraction properties.
-

Homework

- Reading:
 - Chapter 1
 - Familiarize yourself with ML
 - Next time:
 - Mathematical preliminaries
 - Introduction to untyped arithmetic
 - Introduction to Coq
-