

CS 565:

Programming Languages

Spring 2008

Tu, Th: 16:30 - 17:45

Room LWSN 1106

Administrivia

- Who am I?
 - Course web page
 - <http://www.cs.purdue.edu/homes/peugster/CS565Spring08/>
 - Office hours
 - By appointment
 - Main text
 - Types and Programming Languages,
B. Pierce, MIT Press
-

Course Work

- Lectures
 - Homeworks
 - Periodic (probably one every three weeks or so)
 - The answers to 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 CS456 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/page for ML implementations
 - <http://www2.imm.dtu.dk/~riis/PPA/ppasup2004.html>
 - <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, 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, ...)
-

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

Language Design

- Designed to fill a void
 - Enable expression of previously difficult-to-express applications
 - Main overhead
 - Programmer training
 - Languages with many users rarely get replaced (Cobol)
 - Ossification
 - Easy to start in a new niche
-

Language Design

- Tower of Babel
 - Applications often have distinct (and conflicting) needs
 - AI (Lisp, Prolog, Scheme)
 - Scientific computing (Fortran)
 - Business (Cobol)
 - Systems programming (C)
 - Scripting (Perl, Javascript)
 - Distributed computation (Java)
 - Special-purpose (.....)
 - Important to understand differences and similarities among different language features
-

Paradigms

- Imperative
 - Fortran, Algol, C, Pascal
 - Designed around a notion of a program store
 - Program behavior expressed in terms of transformations on the store
 - Functional
 - Lisp, ML, Scheme, Haskell
 - Programs described in terms of a collection of functions
 - "Pure" functional languages are state-free
 - Logic
 - Prolog
 - Programs described in terms of a collection of logical relations
 - Concurrent
 - Fortran90, CSP, Linda
 - Special purpose
 - TeX, Postscript, HTML
-

Metrics

- No universally accepted criteria
 - The most popular languages are not necessarily the best ones
 - Consider Cobol or JCL (Job Control Language)
 - Although, aren't notions of superiority highly subjective?
 - General characteristics
 - Simplicity and "elegance" (orthogonality)
 - Readability
 - Safety
 - Programming-in-the-large
 - Efficiency
 - Abstraction
-

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

- Eiffel
 - Strongly-typed object-oriented language that supports inheritance
 - Uses a notion of covariance (rather than contravariance) in typing functions
 - A function with a restricted set of inputs can be used in a context where a function with a wider set of arguments is expected
 - Type system may fail to prevent runtime type errors
 - Failure to cleanly separate notions of subtyping from subclassing
-

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
-