

CS 565: Programming Languages

Handout 1

January 11, 2009

Instructor

Lecturer

Prof. Suresh Jagannathan
Room 3154J
Lawson Computer Science Building
Ph: x4-0971
suresh@cs.purdue.edu

Office Hours:

Tu, Th 2-3 PM

Course Overview

This course will examine the design and implementation of programming languages from a foundational perspective. Our goal will be to develop tools that will enable us to both design and specify new language features, to precisely understand the rationale for existing features in modern languages, and to understand how design decisions can impact implementations. We will explore ways to understand programming languages that help us to answer questions such as the following:

1. How is feature X found in language Y different from feature X' found in language Y' ? How do we describe its differences precisely?
2. How does incorporating a feature X into a language Y impact the behavior of other constructs in Y ? How do we translate our intuitive notion of behavior into a precise specification?
3. Is including feature X into language Y safe? How do we specify notions of safety?
4. How can we describe feature X in language Y ? How do we precisely distinguish between *essential* and *convenient* language features?
5. What are sensible and tractable notions of program correctness?
6. What modularity techniques can we bring to bear to reason about complexity in programs and languages?
7. How do we qualify the “expressive power” of a language feature?
8. How do we qualify notions of abstraction and types found in modern programming languages? Can we formalize the relationship between types and computation?
9. What is the nature of a *proof* in a programming language?
10. How do we precisely reason about types, values, and other forms of abstraction?
11. How can we understand and use automated software tools that mechanize meta-theory to prove important program properties such as soundness, progress, completeness, etc.?

To help answer these questions, we will develop an unambiguous vocabulary in which to describe the meaning of a language, and will devise various toolkits that use this vocabulary to allow us to build, explore, and formally validate new language features. The vocabulary we use will be predominately drawn from the language of *types*, and we will use variants of the typed λ -calculus as a concrete manifestation. The toolkits we will build will correspond to different type systems. In addition, we will explore different formalisms to describe the dynamic behavior of programs and will relate *static* safety properties as defined by a type system with *dynamic* notions of correctness as defined by a language semantics. We will also examine formal ways to abstract the execution properties of a program using principles drawn from abstract interpretation. We will employ proof assistant tools that help mechanize proofs of correctness to validate important features of these systems.

At the end of this class, students should be comfortable with modern notions of language specification, have developed sufficient mathematical maturity to formally prove (both manually and through tools) interesting properties about a specific language feature or type system, and be capable of understanding current research trends in the field.

The course will be divided into roughly four parts:

1. The first part of the course will provide an overview of different semantic models, will introduce notions of interpretation, will expose students to important mathematical devices critical to reasoning about programs such as structural induction, and will include an in-depth study of the *untyped* λ -calculus.
2. The second part of the course will introduce types, motivate their utility as a lightweight formal specification in which diverse language features can be examined, and will introduce simple type systems in the context of *typed* λ -calculus.
3. The third part of the class will explore more advanced concepts including subtyping and inheritance in object-oriented languages, recursive data structures, existential types for data abstraction, and polymorphism. Time permitting, a brief introduction to topics such as linear types, theorem proving, and types in concurrency will be given.
4. The fourth part of the course will introduce students to mechanized proof assistants. We will use Coq, a proof assistant that allows for the interactive construction of formal proofs.

Prerequisites

It is assumed that students taking this class would have had some exposure to an undergraduate programming languages class (equivalent to CS 465), and are comfortable with basic mathematical concepts (e.g., graph and set theory). Although students need not be fluent in ML (the language in which sample programs in the course text are given, and in which programming projects must be implemented), they should be confident in their ability to grasp its key features by self-study.

Programming Assignments

There will be a number of small programming exercises given during the course. These exercises will require students to implement type systems and interpreters of varying complexity and sophistication. The implementations will be written in ML. Students are free to discuss all aspects of the exercise with other classmates, but must write and submit their programs individually.

Academic Honesty

Students are encouraged to collaborate on homeworks, and to work together to clarify issues presented in class, etc. Answers to most of the questions given in the homework assignments will be available in the back of the text. Students should thus approach homework assignments with an honest spirit of inquiry; simply copying the answers from the text without full understanding is of no benefit.

Grading

Grading for this class will be based as follows:

Midterm	30%
Final	40%
Programming Assignments	30%

The final examination will also serve as the Ph.D qualifying examination for this subject.

Texts

The course will mostly follow **Types and Programming Languages**, by Benjamin C. Pierce (MIT Press 2002). Some material will also be drawn from **Principles of Program Analysis**, by Nielson, Nielson, and Hankin (Springer, 2005).

In addition, students might also find the following texts useful:

- **Formal Semantics**, Glynn Winskel, MIT Press 1993.
- **Essentials of Programming Languages**, Daniel Frideman, Mitchell Wand, and Christopher Haynes, MIT Press 2001.
- **Foundations of Programming Languages**, John Mitchell, MIT Press, 1996.
- **The Structure of Typed Programming Languages**, David Schmidt, MIT Press, 1994.

Resources

Since the course text uses Caml (a dialect of ML) for programming exercises and sample program descriptions, the Caml homepage (at <http://caml.inria.fr>) is an excellent resource. At this site, you will find tutorials, technical descriptions, sample applications, etc. Other useful resources include:

1. Details about Standard ML of New Jersey (SML/NJ), an SML implementation supported on the department servers can be accessed at <http://www.smlnj.org>.
2. The SML basis library contains signatures for a number of SML structures that are part of all Standard ML implementations. It can be found at: <http://www.standardml.org/Basis>.

3. Robert Harper at CMU has extensive notes on programming in Standard ML; you can find it at www-2.cs.cmu.edu/~rwh/smlbook/online.pdf.
4. Peter Lee has some useful notes on using SML/NJ at: <http://www-2.cs.cmu.edu/~petel/smlguide/smlnj.htm>.
5. Alice is an implementation of SML with extensions for concurrent and distributed programming. Details can be found at <http://www.ps.uni-sb.de/alice>.
6. MLton is a whole program optimizing compiler for Standard ML. Details about MLton can be found at <http://www.mlton.org>. An implementation of MLton is also available on the departmental machines. There are also numerous links to SML resources and tutorials on aspects of using SML from: www.mlton.org/StandardML.

Programming assignments will be in Standard ML. The implementation of SML supported on the department servers is Standard ML of New Jersey (`smlnj`), but you are free to use any other implementation of SML if you wish.

You are encouraged to review past proceedings of the ACM Principles of Programming Languages (POPL), ACM Programming Language Design and Implementation (PLDI), and the ACM International Conference on Functional Programming (ICFP). These conferences all publish articles related to programming language design and theory.

Web Pages and Newsgroups

Copies of the lectures and handouts will be available through the course web page: <http://www.cs.purdue.edu/homes/suresh/565-Spring2009/index.html>.

There is a CS mailing list: `cs565-spring2009@cs.purdue.edu` that we'll use to broadcast important messages (e.g., errata) about the lectures, homeworks, and programming assignments. You should add yourself to this list using the command: `mailer add me to cs565-spring2009`.

There is also a web blog at <http://cs565spring2009.wordpress.com> that can be used as a general question and discussion forum site. Students are free to post responses to any questions posed.

Tentative Syllabus

The course will roughly follow the breakdown given below. The midterm will cover material upto and including simply-typed lambda-calculus, but may subsume more depending upon the rate at which the lectures proceed. The midterm will be given in the week immediately preceding spring break.

<i>Lectures</i>	<i>Topics</i>	<i>Text Reference</i>
1 – 15	Introduction. Untyped Arithmetic Operational Semantics Proof Techniques Domain Theory and Fixpoints Axiomatic Semantics Abstract Interpretation Lambda-Calculus	Chap. 1 Chaps. 2 & 3 Chap. 3 Notes and Chap. 4 Winskel Winskel Nielson Chaps. 5 & 6
16 – 32	Introduction to Types Simply-Typed Lambda Calculus Simple Extensions References and Exceptions Subtyping Imperative Objects Featherweight Java Recursive Types	Chap. 8 Chap. 9 Chap. 11 Chaps. 13 & 14 Chaps. 15 & 16 Chap. 18 Chap. 19 Chap. 20
33 – 45	Polymorphism Universal Types, System F Existential Types Bounded Quantification	Chap. 22 Chap. 23 Chap. 24 Chap. 26

Time permitting, we will also discuss other advanced topics including contracts, hybrid typechecking, linear types, regions, automated theorem proving, types for security, and concurrency. The additional material covered will depend upon class interest.