CS258. Introduction to Programming Language Theory

This course studies a family of “model programming languages,” their fundamental properties, and some relationships to programming practice.

Basic approach: Typed lambda calculus, augmented with
- Basic data types (integers, Booleans, ...)
- Recursion, and variants such as primitive recursion
- Assignable locations
- Polymorphism, data abstraction, modules... (although the course may not cover these...)

Compare to: Finite automaton, augmented with
- Pushdown store (i.e., pushdown automata)
- Infinite tape (i.e., Turing machine)
- Oracle tape
- Randomization or nondeterminism

The theory of programming languages based on typed lambda calculus is analogous to the theory of computation based on varieties of automata and related machine models. In both cases, the language or execution model is not exactly the same as the ones we use in practice, but these simplified models help us focus on central issues and prove results rigorously.
Outline

1. Functional programming and typed lambda calculus (Chapter 2)
   a. Boolean, natural number, pairing and function expressions; definition of recursive functions using fixed-point operator (Section 2.2)
   b. Comparison of axiomatic, operational and denotational semantics (Section 2.3)
   c. Properties of reduction; deterministic symbolic interpreters (Section 2.4)
   d. Programming techniques, expressive power, limitations (Section 2.5)

2. Universal algebra and algebraic data types (Chapter 3)
   a. Algebraic terms, equations and algebras (Sections 3.1-3)
   b. Equational proof system, soundness and completeness (Section 3.4)
   c. Homomorphisms and initiality (Section 3.5)
   d. Aspects of algebraic theory of data types (Section 3.6)

3. Semantics of typed lambda calculus and recursion (Parts of Chapters 4 and 5)
   a. Presentation of context-sensitive syntax by typing rules (Sections 4.3.1, 4.3.2, 4.3.5)
   b. General models, summary of soundness and completeness (Sections 4.5.1-4)
   c. Domain-theoretic models of typed lambda calculus with fixed-point operators (Sections 5.1 and 5.2; Sections 5.3 and 5.4 time permitting)

4. Imperative programs (Chapter 6)
   a. Syntax of while programs; L-values and R-values (Section 6.2)
   b. Structured operational semantics (Section 6.3)
   c. Denotational semantics using typed lambda calculus with location and store types, fixed-point operator (Section 6.4)
   d. Partial correctness assertions. Soundness, relative completeness and example proofs (Section 6.5).
**Typed Lambda Calculus**

Basic language with
- Variable binding
- Function definitions
- Function application (function “calls”)
- Types

There is also an “untyped” lambda calculus, in which variables and expressions are not given a type. However, untyped lambda calculus is a special case of typed lambda calculus.

*Idea:* there is a universal type U, and every type needed is included in U.

*Example:* datatype untyped = N of int | B of bool | P of untyped*untyped | F of untyped → untyped

*Note:* this is a recursively defined type, which we will not get to in the early part of this course.

**Lambda Notation: Abstraction and Application**

If $M: \tau$ contains a variable $x: \sigma$ then $\lambda x: \sigma. M$ has type $\sigma \rightarrow \tau$

Typing rule (avoided in early chapters of book but better to cover now)

$$
\frac{\tau \vdash M: \tau}{x: \sigma \vdash \lambda x: \sigma. M : \sigma \rightarrow \tau}
$$

Examples:

1) $x+5 : nat$ contains variable $x:nat \implies \lambda x:nat. x+5 : nat \rightarrow nat$

2) if b then “hot” else “cold” : string contains variable $b:bool \implies \lambda b:bool. (...) : bool \rightarrow string$

If $M: \sigma \rightarrow \tau$ and $N: \sigma$ then $MN: \tau$

Typing rule

$$
\frac{M: \sigma \rightarrow \tau \quad N: \sigma}{MN: \tau}
$$

**Lambda Calculus**

Equational principal: Alpha-equivalence

Function application: Beta-equivalence, beta-reduction

Confluence: “order doesn’t matter”, reduction is parallelizable

Termination: evaluation halts / typed lambda calculus (so far) is not a universal programming language
**Course Goals**
Understand programming language concepts precisely and provably.

Understand meaning in different ways

- Operational
- Denotational
- Axiomatic

Understand and prove basic theorems about illustrative systems, to appreciate them and to learn how to prove other theorems as needed for other purposes.

**Background**
Syntax: grammar, parse trees, abstract syntax (expressions of a language are its parse trees)

Induction on expressions

**Reading for next class (Chapter 1 of textbook)**
Read: Sections 1.1-3 and Sections 1.7-8, up through induction on expressions. Skim section covering induction on proofs so you know it is there when we need it later in the course.