Theory of Types and Programming Languages

Lionel PARREAUX, HKUST

Slides in part adapted from:

University of Pennsylvania CIS 500: Software Foundations – Fall 2006
by Benjamin Pierce

EPFL CS-452: Foundations of software – Fall 2021
by Martin Odersky

Course Overview

What is "TTAPL"?

The theory of programming languages is the mathematical study of the meaning of programs.

The goal is to describe program behaviors in ways that are both precise and abstract.

- precise so that we can use mathematical tools to formalize and check interesting properties
- abstract so that properties of interest can be discussed clearly, without getting bogged down in low-level details

Why study the theory of programming languages?

- To prove specific properties of particular programs (program verification)
  - Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive
- To develop intuitions for *informal* reasoning about programs
- To prove general facts about all the programs in a given programming language (e.g., safety or isolation properties)
- To understand language features (and their interactions) deeply and develop principles for better language design (PL is the "materials science" of computer science...)

What you can expect to get out of the course

- A more sophisticated perspective on programs, programming languages, and the activity of programming
  - See programs and whole languages as formal, mathematical objects
  - Make and prove rigorous claims about them
  - Have detailed knowledge of a variety of core language features
- Deep intuitions about key language properties such as type safety
- Powerful tools for language design, description, and analysis

Most software designers are language designers!

Greenspun's Tenth Rule Of Programming

"Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp."

– Philip Greenspun

What this course is not

- An introduction to programming
- A course on functional programming (though we'll be doing some functional programming along the way)
- A course on compilers (you should already be comfortable with basic concepts such as lexical analysis, parsing, abstract syntax, and scope)
- A comparative survey of many different programming languages and styles

Approaches to Program Meaning

- Denotational semantics and domain theory view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
- Program logics such as Hoare logic and dependent type theories focus on logical rules for reasoning about programs.
- Operational semantics describes program behaviors by means of abstract machines. This approach is somewhat lower-level than the others, but is extremely flexible.
- Process calculi focus on the communication and synchronization behaviors of complex concurrent systems.
- Type systems describe approximations of program behaviors, concentrating on the shapes of the values passed between different parts of the program.

Overview (tentative)

This course will concentrate on operational techniques and type systems.

- Part I: Modeling programming languages
  - Syntax and parsing
  - Operational semantics
  - Inductive proof techniques
  - The lambda calculus
  - Syntactic sugar; fully abstract translations
- Part II: Type systems
  - Simple types
  - Type safety
  - Recursion, State, and Other Extensions
  - Polymorphism and Type Reconstruction
  - Subtyping

Overview

- Part III: Advanced Types
  - Structural and Object-Oriented Features
  - Dependent Types
  - Curry-Howard Correspondence
  - Foundations of Scala's Type System

Organization of the Course

Staff

Instructor:	Lionel parreaux@cse.ust.hk
Teaching Assistant:	Luyu Cheng luyu.cheng@connect.ust.hk

Information

Textbook: Types and Programming Languages,
Benjamin C. Pierce, MIT Press, 2002

Webpage: `https://canvas.ust.hk/courses/42319/`

Other relevant resources (for deeper investigation):

- Advanced topics in types and programming languages. B. C. Pierce, MIT press, 2005.
- Software foundations, B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, & B. Yorgey, 2010.

Elements of the Course

- The TTAPL course consists of
  - lectures (Wed 16:30-17:50, on Zoom)
  - exercises and project work (Fri 16:30-17:50, on Zoom)

  (real-time online mode until end of add-drop period)
- The lecture will follow in part the textbook.
- For lack of time, we cannot treat all essential parts of the book in the lectures. That's why the textbook is required reading for participants of the course.

Homework and Projects

You will be asked to

- solve and hand in some written exercise sheets,
- do a number of programming assignments, including
  - interpreters and reduction engines
  - type checkers

  for a variety of small languages.
- The recommended implementation language for these assignments is Scala.

Scala

- Scala is a functional and object-oriented language that is closely interoperable with Java.
- It is very well suited as an implementation language for type-checkers, in particular because it supports:
  - pattern matching,
  - higher-order functions,
  - an expressive object system.

Learning Scala

If you don't know Scala yet, there's help:

- The Scala web site:

  www.scala-lang.org

- On this site, the documents:
  - *A Brief Scala Tutorial - an introduction to Scala for Java programmers.* (short and basic).
  - *An Introduction to Scala* (longer and more comprehensive).
  - *An Overview of the Scala Programming Language* (high-level).
  - *Scala By Example* (long, comprehensive, tutorial style).
- The assistants.

Grading and Exams

Final course grades will be computed as follows:

- Homework and project: 30%
- Mid-term exam: 30%
- Final exam: 40%

Collaboration

- Collaboration on homework is strongly encouraged.
- Studying with other people is the best way to internalize the material
- Form study groups!

"You never really misunderstand something
until you try to teach it...
" – Anon.

Plagiarism

While collaboration on exercise homework is encouraged,
the projects and exams are individual.

Plagiarizing code written by other people as part of a project is unethical and will not be tolerated, whatever the source.

Part I

Modelling programming languages

Syntax and Parsing

- The first-level of modeling a programming language concerns its context-free syntax.
- Context free syntax determines a set of legal phrases and determines the (tree-)structure of each of them.
- It is often given on two levels:
  - concrete: determines the exact (character-by-character) set of legal phrases
  - abstract: concentrates on the tree-structure of legal phrases.
- We will be mostly concerned with abstract syntax in this course.
- But to be able to write complete programming tools, we need a convenient way to map character sequences to trees.

Approaches to Parsing

There are two ways to construct a parser:

- By hand Derive a parser program from a grammar.
- Automatic Submit a grammar to a tool which generates the parser program.

In the second approach, one uses a special grammar description language to describe the input grammar.

Domain-Specific Languages

- The grammar description language is an example of a domain-specific language (DSL).
- The parser generator acts as a processor ("compiler") for this language — that's why it's sometimes called grandly a "compiler-compiler".
- Example of a "program" in the grammar description DSL: (Grammar A)

```
Expr   ::= Term {'+' Term | '−' Term}.
Term   ::= Factor {'∗' Factor | '/' Factor}.
Factor ::= Number | '(' Expr ')'.
```

Embedded Domain Specific Languages

- An alternative to a stand-alone DSL is an Embedded DSL.
- Here, the DSL does not exist as a separate language but as an API in a host language.
- The host language is usually a general purpose programming language.

In this course, we use a Scala Embedded DSL to do the parsing.

An EDSL for Parsing in Scala

Here is what it looks like:

(Grammar B)

```
def expr   : Parser[Any] = term ~ rep("+" ~ term | "-" ~ term)
def term   : Parser[Any] = factor ~ rep("*" ~ factor | "/" ~ factor)
def factor : Parser[Any] = "(" ~ expr ~ ")" | numericLit
```

This course is not about parsing.

We will provide the parser implementations to you.

Parser Combinators

- The differences between Grammar A and Grammar B are fairly minor.

  (Note in particular that existing DSLs for grammar descriptions also tend to add syntactic complications to the idealized Grammar A we have seen).

- The important difference is that Grammar B is a valid Scala program, when combined with an API that defines the necessary primitives.

- These primitives are called parser combinators.

Concrete and Abstract Syntax

Concrete grammar, for parsing:

```
Expr   ::= Term {'+' Term | '−' Term}.
Term   ::= Factor {'∗' Factor | '/' Factor}.
Factor ::= Number | '(' Expr ')'.
```

What we really care about: the abstract "grammar" or abstract syntax

```
/** Abstract Syntax Trees for terms. */

enum Term:
  case NumericLit(value: Int)
  case Binop(lhs: Term, op: Operator, rhs: Term)

enum Operator:
  case Plus
  case Minus
  case Times
  case Div
```

To Prepare For Next Lecture

You should try to at least look at the reading material for a particular lecture before that lecture.

Next week, we'll start with Chapter 3 of the textbook.

(Chapter 2 contains some mathematical preliminaries which we assume you are familiar with.)