编程语言的设计原理
Design Principles of Programming Languages

Haiyan Zhao, Di Wang

赵海燕，王迪

Peking University, Spring Term 2023

Teaching Team

Instructors

Teaching Assistant

Instructors

- Haiyan Zhao （赵海燕）
  - 1988, BS, Peking Univ.
  - 1991, MS, Peking Univ.
  - 2003, PhD, Univ. of Tokyo
  - 2003-, Assoc. Professor, Peking Univ.
- Research Interest
  - Software engineering
  - Requirements Engineering, Domain Engineering
  - Programming Languages
- Contact
  - Office: Rm. 1809, Science Blg #1, Yanyuan / Rm 432, CS Blg, Changping
  - Email： zhhy@sei.pku.edu.cn
  - Phone： 62757670

Instructors

- Di Wang （王迪）
  - 2017, BS, Peking Univ.
  - 2022, PhD, Carnegie Mellon Univ.
  - 2022-, Assistant Professor, Peking Univ.

- Research Interests
  - Programming Languages
  - Quantitative Program Analysis and Verification
  - Probabilistic Programming

- Contact
  - Office: Rm. 520, Yanyuan Mansion
  - Tel: 62757242
  - Email: wangdi95@pku.edu.cn
  - Webpage: https://stonebuddha.github.io

Teaching Assistant

- Lijuan Tang （唐丽娟）
  - PhD student，Programming Languages Laboratory
  - Email：lijuan_tang@stu.pku.edu.cn

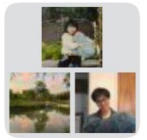
Information

- Time： Monday 7-9 (15:10-18:00)
- Place： 昌平教学楼 201

- Course website:

  https://pku-dppl.github.io/2023/

  – Syllabus
  – News/Announcements
  – Lecture Notes (slides)
  – Other useful resources



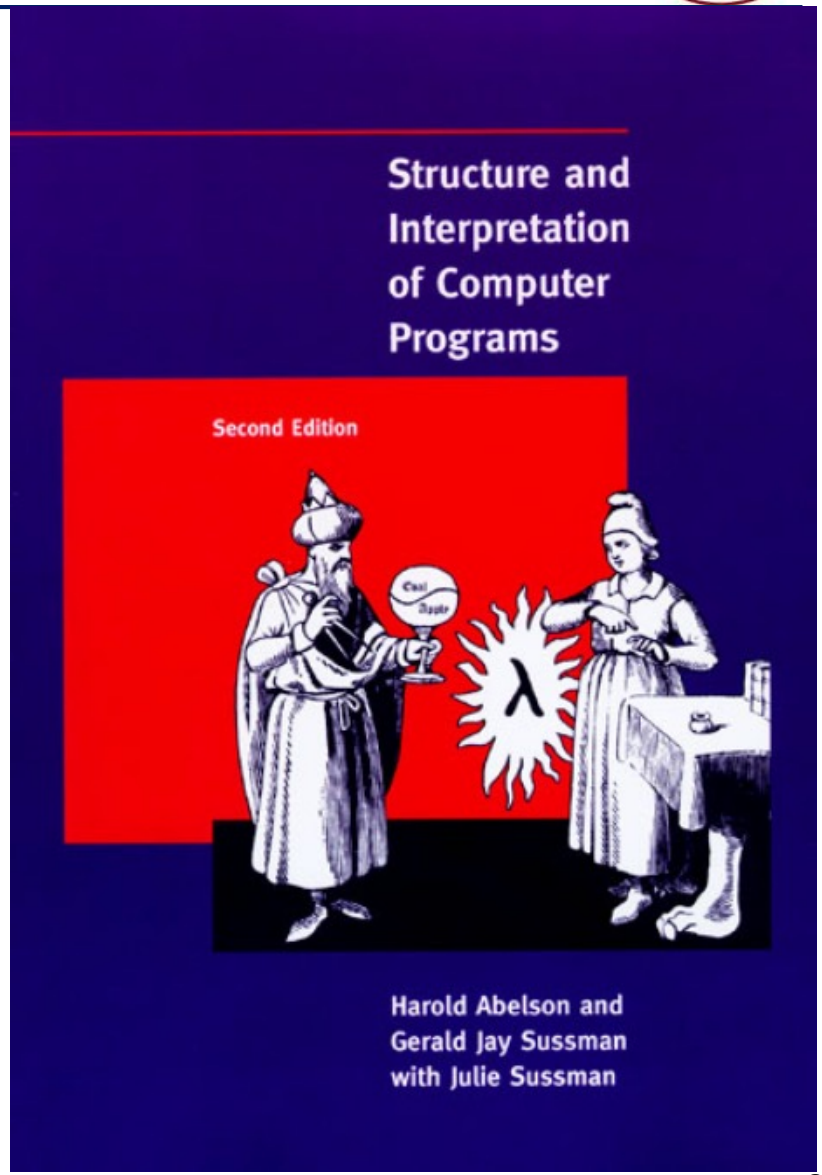
群聊: DPPL-2023春季课程群

该二维码7天内(2月25日前)有效，重新进入将更新

Course Overview

Computer Science = PL Construction

" . . . the technology for coping with *large-scale computer systems* merges with the technology for building *new computer languages*, and *computer science itself* becomes no more (and no less) than the discipline of *constructing appropriate descriptive languages*"



What is this course about?

- Study fundamental (formal) approaches to describing *program behaviors* 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

What you can get out of this course?

- A more *sophisticated perspective* on programs, programming languages, and the activity of programming
  - How to *view programs and whole languages* as formal, mathematical objects
  - How to *make and prove rigorous claims* about them
  - Detailed *study* of a range of *basic language features*

- Powerful tools/techniques for language design, description, and analysis

This course is not about …

- An introduction to programming
- A course on compiler
- A course on functional programming
- A course on language paradigms/styles

All the above are certainly helpful for your deep understanding of this course.

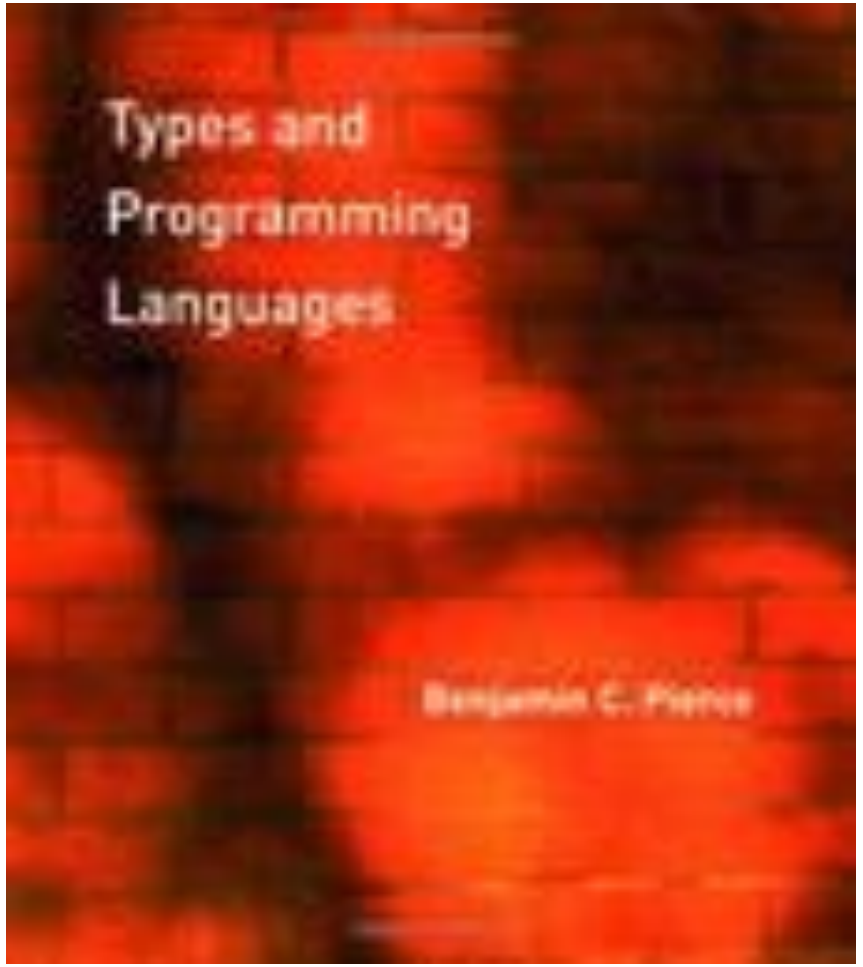
What background is required?

- Basic knowledge on
  - Discrete mathematics: sets, functions, relations, orders
  - Algorithms: list, tree, graph, stack, queue, heap
  - Elementary logics: propositional logic, first-order logic

- Familiar with a *programming language* and basic knowledge of *compiler construction*

Textbook

- Types and Programming Languages
  - Benjamin Pierce
  - The MIT Press
  - 2002-02-01
  - ISBN 9780262162098



Outline

- Basic operational semantics and proof techniques
- Untyped Lambda calculus
- Simple typed Lambda calculus
- Simple extensions (basic and derived types)
- References
- Exceptions
- Subtyping
- Recursive types
- Polymorphism
- [Higher-order systems]

Grading

- Activity in class + midTest : 20%
- Homework: 40%
- Final (Report/Presentation): 40%

设计一个带类型系统的程序语言，解决实践中的问题，给出基本实现

- 设计一个语言，保证永远不会发生内存/资源泄露。
- 设计一个汇编语言的类型系统
- 设计一个没有停机问题的编程语言
- 设计一个嵌入复杂度表示的类型系统，
  保证编写的程序的复杂度不会高于类型标示的复杂度。
- 设计一个类型系统，使得敏感信息永远不会泄露。
- 设计一个类型系统，使得写出的并行程序没有竞争问题
- 设计一个类型系统，保证所有的浮点计算都满足一定精度要求
- 解决自己研究领域的具体问题

How to study this course?

- Before class: scanning through the chapters to learn and gain feeling about what will be studied
- In class: trying your best to understand the contents and raising hands when you have questions
- After class: doing exercises seriously

★	Quick check	30 seconds to 5 minutes
★★	Easy	≤ 1 hour
★★★	Moderate	≤ 3 hours
★★★★	Challenging	> 3 hours

Chapter 1: Introduction

What is a type system

What type systems are good for

Type systems and programming languages

Types in PL (CS)

1870s	*origins of formal logic*	Frege (1879)
1900s	*formalization of mathematics*	Whitehead and Russell (1910)
1930s	*untyped lambda-calculus*	Church (1941)
1940s	*simply typed lambda-calculus*	Church (1940), Curry and Feys (1958)
1950s	Fortran	Backus (1981)
	Algol-60	Naur et al. (1963)
1960s	*Automath project*	de Bruijn (1980)
	Simula	Birtwistle et al. (1979)
	Curry-Howard correspondence	Howard (1980)
	Algol-68	(van Wijngaarden et al., 1975)
1970s	Pascal	Wirth (1971)
	Martin-Löf type theory	Martin-Löf (1973, 1982)
	System F, F^{ω}	Girard (1972)
	polymorphic lambda-calculus	Reynolds (1974)
	CLU	Liskov et al. (1981)
	polymorphic type inference	Milner (1978), Damas and Milner (1982)
	ML	Gordon, Milner, and Wadsworth (1979)
	intersection types	Coppo and Dezani (1978)
		Coppo, Dezani, and Sallé (1979), Pottinger (1980)
1980s	NuPRL project	Constable et al. (1986)
	subtyping	Reynolds (1980), Cardelli (1984), Mitchell (1984a)
	ADTs as existential types	Mitchell and Plotkin (1988)
	calculus of constructions	Coquand (1985), Coquand and Huet (1988)
	linear logic	Girard (1987) , Girard et al. (1989)
	bounded quantification	Cardelli and Wegner (1985)
		Curien and Ghelli (1992), Cardelli et al. (1994)
	Edinburgh Logical Framework	Harper, Honsell, and Plotkin (1992)
	Forsythe	Reynolds (1988)
	pure type systems	Terlouw (1989), Berardi (1988), Barendregt (1991)
	dependent types and modularity	Burstall and Lampson (1984), MacQueen (1986)
	Quest	Cardelli (1991)
	effect systems	Gifford et al. (1987), Talpin and Jouvelot (1992)
	row variables; extensible records	Wand (1987), Rémy (1989)
		Cardelli and Mitchell (1991)
1990s	higher-order subtyping	Cardelli (1990), Cardelli and Longo (1991)
	typed intermediate languages	Tarditi, Morrisett, et al. (1996)
	object calculus	Abadi and Cardelli (1996)
	translucent types and modularity	Harper and Lillibridge (1994), Leroy (1994)
	typed assembly language	Morrisett et al. (1998)

What is a type system (type theory)?

- A *type system* is a tractable syntactic method for proving the *absence of certain program behaviors* by classifying phrases according to the kinds of values they compute.
  - Tools for program reasoning*
  - Classification of terms
  - Static approximation**
  - Proving the absence rather than presence (conservative)
  - Fully automatic (and efficient)

What are type systems good for?

- Detecting Errors
  - Many programming errors can be detected early, fixed intermediately and easily.
  - Errors can often be pinpointed more accurately during typechecking than at run time, when their effects may not become visible until some time after things begin to go wrong.
  - Expressive type systems offer numerous “tricks” for encoding information about structure in terms of types
- Abstraction
  - Type systems form the backbone of the module languages and tie together the components of large systemsin the context of large-scale software composition
  - an interface itself can be viewed as “the type of a module” , providing a summary of the facilities provided by the module
- Documentation
  - The type declarations in procedure headers and module interfaces constitute a form of (checkable) documentation.
  - this form of documentation cannot become outdated as it is checked during every run of the compiler. This role of types is particularly important in module signatures.

What are type systems good for?

- Language Safety
  - A safe language is one that protects its own abstractions.
  - Safety refers to the language's ability to guarantee the integrity of these abstractions and of higher-level abstractions introduced by the programmer using the definitional facilities of the language.
  - Language safety is not the same thing as static type safety, and can be achieved by static checking, but also by run-time checks
- Efficiency
  - Removal of dynamic checking; smart code-generation
  - most high-performance compilers rely heavily on information gathered by the typechecker during optimization and code-generation phases.

Type Systems and Languages Design

- Language design should go hand-in-hand with type system design.
  - Languages without type systems tend to offer features that make *typechecking difficult or infeasible*.
  - Concrete syntax of typed languages tends to be *more complicated* than that of untyped languages, since type annotations must be taken into account.

> In typed languages the type system itself is often taken as the foundation of the design and the organizing principle in light of which every other aspect of the design is considered.

Homework

- Read Chapters 1 and 2.

- Install OCaml and read “Basics”
  - Overview
    - https://ocaml.org/docs/
  - Installation
    - https://ocaml.org/docs/up-and-running