Welcome to the course!

Advanced Compiler Construction and Program Analysis

Course introduction

Innopolis University, Spring 2022

What is this course about?

- Primary focus is on **type systems**
  - Design and implementation
  - Properties and trade-offs

- Secondary focus is on **compilation** and run-time support for **lazy programming languages**

What is a type system?

> *“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.”*
>
> Benjamin C. Pierce

What is a type system?

> *"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."*

Benjamin C. Pierce

Type system is a tool
for reasoning about **programs**!

What is a type system?

> *“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.”*

Benjamin C. Pierce

A type system helps calculate a kind of **static** approximation to the run-time behaviour.

Static Type Systems are Conservative

Consider the following example:

```
if <complex test> then 5 else <type error>
```

A static type system will (most likely) reject this program as *ill-typed* even if `<complex test>` will always evaluate to `true`.

What errors can/should a type system catch?

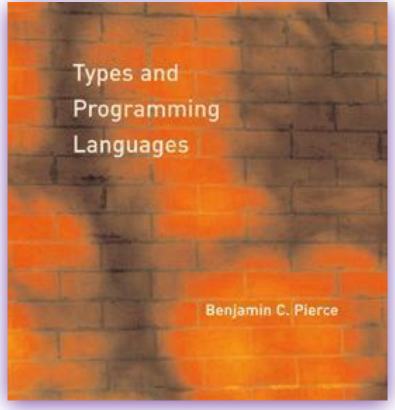
- Bad behaviours that can be eliminated by the type system are sometimes called ***run-time type errors***.
- The ***safety*** (or ***soundness***) of each type system must be judged with respect to its own set of run-time errors.
- Type systems also can enforce higher-level ***modularity*** properties and protect the integrity of user-defined ***abstractions***.
- Type checkers (in programming languages) are usually ***automatic***, requiring no manual intervention*.

What are Type Systems good for?

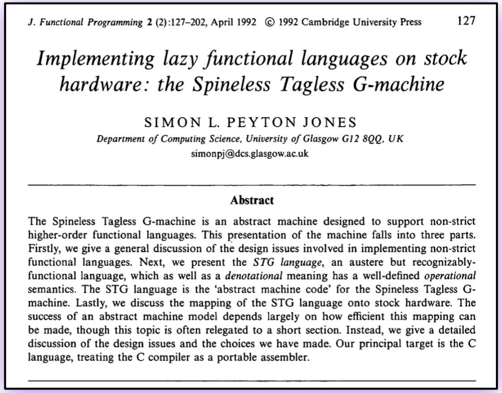
- Detecting Errors
- Abstractions
- Providing (limited) documentation
- Language Safety
  (e.g. purity in Haskell, memory management in Rust)
- Efficiency (e.g. type-assisted optimizations)
- And more (static analysis, network security, theorem provers, database systems, etc.)

Materials for this course

- Benjamin C. Pierce.
  Types and Programming Languages
  MIT Press 2002



- Simon Peyton Jones.
  Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine.
  Journal of Functional Programming 1992



J. Functional Programming **2** (2):127–202, April 1992 © 1992 Cambridge University Press 127

Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine

SIMON L. PEYTON JONES
Department of Computing Science, University of Glasgow G12 8QQ, UK
simonpj@dcs.glasgow.ac.uk

Abstract

The Spineless Tagless G-machine is an abstract machine designed to support non-strict higher-order functional languages. This presentation of the machine falls into three parts. Firstly, we give a general discussion of the design issues involved in implementing non-strict functional languages. Next, we present the *STG language*, an austere but recognizably-functional language, which as well as a *denotational* meaning has a well-defined *operational* semantics. The STG language is the 'abstract machine code' for the Spineless Tagless G-machine. Lastly, we discuss the mapping of the STG language onto stock hardware. The success of an abstract machine model depends largely on how efficient this mapping can be made, though this topic is often relegated to a short section. Instead, we give a detailed discussion of the design issues and the choices we have made. Our principal target is the C language, treating the C compiler as a portable assembler.

Topics, covered in this course

❑ Simply Typed λ-calculus

❑ Subtyping and Imperative Objects

❑ Universal Types, System F, Hindley-Milner type system

❑ Implementing lazy languages. STG-machin

Course structure

- Lectures
  - Provide necessary **theoretical material**
  - May include **tests/quizzes**

- Labs
  - **Live coding** and **analysis** of implementations of different features of programming languages
  - Intermediate **project presentations**

- Team projects
  - **Design** of a custom typed programming language
  - **Implementation** of an interpreter and a type checker
  - **Documentation** and presentation

Grading structure

❖ Tests during lectures	— 10%
❖ Lab participation	— 20%
❖ Projects:	
➢ Language design	— 10%
➢ Interpreter/compiler	— 20%
➢ Type checker	— 20%
➢ Tests & Documentation	— 20%

Grading policy

A	>90%
B	>75%
C	≥60%
D	<60%