Advanced Compiler Construction and Program Analysis

- Course name: Advanced Compiler Construction and Program Analysis
- Course number: n/a
- Subject area: Programming Languages and Software Engineering

Course characteristics

Key concepts of the class

- Type Systems
- Lambda calculi as the core representation
- Type checking and type inference
- Simple types and derived forms
- Subtyping
- Imperative objects
- Recursive types
- Universal polymorphism
- Compiling lazy functional languages

What is the purpose of this course?

This course partially covers two major topics:

1. Theory and Implementation of Typed Programming Languages and
2. Compilation of Lazy Functional Languages.

We will study different type system features in detail, starting from an untyped language of lambda calculus and gradually adding new types and variations along the way. The course assumes familiarity with basics of compiler construction, basics of functional programming and familiarity with some static type systems (C++ and Java would suffice, but knowing type systems of Haskell or Scala will help).

Even though the most obvious benefit of static type systems is that it allows programmers to detect some errors early, it is by far not the only application. Types are used also as a tool for abstraction, documentation, language safety, efficiency and more. In this course we will look at features of type systems occurring in many programming languages, like C++, Java, Scala and Haskell.

After we have reached System F, a type system at the core of languages like Haskell, we will look into how lazy functional languages are implemented. We will in particular look in detail at Spineless Tagless Graph reduction machine (also known as STG machine) that is used to compile Haskell code.

Evaluation of the course consists of Lecture Quizzes, Lab Participation and the Final Project (split into several stages). The Final Project is a team project

where students build a complete interpreter or compiler for a statically typed programming language, incorporating some of the features covered in this course.

Course Objectives Based on Bloom's Taxonomy

What should a student remember at the end of the course?

- Remember syntax and computation rules of untyped lambda calculus.
- Remember nameless representation of lambda terms.
- Remember definition of normal form, weak head normal form.
- Remember syntax, typing and computation rules of simply typed lambda calculus.
- Remember definition of imperative objects.
- Remember syntax and semantics of Featherweight Java.
- Remember typing rules for subtyping.
- Remember typing rules for pairs, tuples, records, sums, variants, and lists.
- Remember typing rules for let-bindings and type ascription.
- Remember typing rules for recursive types.
- Remember definition and typing rules for universal polymorphism.
- Remember syntax, typing and computation rules for System F.
- Remember representation of closures when compiling functional languages.
- Remember representation of lazy data structures when compiling.
- Remember the syntax and semantics of the STG language.

What should a student be able to understand at the end of the course?

- Understand how type systems relate to language design.
- Understand differences between call-by-name, call-by-need, and call-by-value evaluation strategies.
- Understand the tradeoffs introduced by various type system features.
- Understand the idea of nameless representation of terms.
- Understand the tradeoffs of mutable references and exceptions introduced in a language.
- Understand how imperative objects model objects in modern object-oriented langauges.
- Understand the difficulties of compiling lazy expressions.
- Understand the differences between Hindley–Milner type system and System F.

What should a student be able to apply at the end of the course?

- Implement an interpreter for a programming language with untyped lambda calculus as its core representation.
- Implement an interpreter for a programming language with simply typed lambda calculus as its core representation.
- Implement type checking algorithm for a language with simple types, recursive types, imperative objects, and universal polymorphism.

- Implement Damas–Hindley–Milner type inference algorithm for a programming language with a Hindley-Milner style type system.

Course evaluation

	Proposed points
Labs/seminar classes	20
Interim performance assessment	10
Exams	70

Labs/seminar classes In-class participation 1 point for each individual contribution in a class but not more than 1 point per class (i.e. 14 points in total for 14 classes), overall course contribution (to accumulate extra-class activities valuable to the course progress, e.g. a short presentation, book review, very active in-class participation, etc.) up to 6 points.

Interim performance assessment In-class tests up to 10 points for each test, computational practicum assignment up to 10 points for each task.

Exams Mid-term exam up to 20 points, final examination up to 30 points.

’‘’Overall score:” ’ 100 points (100%).

Grades range

	Proposed range
A. Excellent	85–100
B. Good	75–84
C. Satisfactory	60–74
D. Poor	0–59

Resources and reference material

- Benjamin C. Pierce. ’‘Types and Programming Languages. The MIT Press 2002”
- Simon Peyton Jones. ’‘Implementing functional languages: a tutorial. Prentice Hall 1992”
- Simon Peyton Jones. ’‘Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. Journal of Functional Programming 1992”

Course Sections

The main sections of the course and approximate hour distribution between them is as follows:

Section	Section Title	Lecture Hours	Seminars (labs)	Self-study	Knowledge evaluation
1	Lambda Calculus and Simple Types	10	10	10	2
2	References, Exceptions, Imperative Objects, Featheweight Java	8	8	8	1
3	Recursive Types, Type Reconstruction, Universal Polymorphism	6	6	6	1
4	Compiling Lazy Functional Languages	8	8	8	1
5	Project Presentation				2

Section 1

Section title:* Lambda calculus and simple types

Topics covered in this section:

- The history of typed languages. Type systems and language design.
- Basic notions: untyped lambda calculus, nameless representation, simple types.

What forms of evaluation were used to test students' performance in this section?

Form of evaluation	Usage
Development of individual parts of software product code	1
Homework and group projects	1
Midterm evaluation	0
Testing (written or computer based)	0
Reports	1
Essays	0
Oral polls	1
Discussions	1

Typical questions for ongoing performance evaluation within this section

- What is the role of the type system in language design?
- How to evaluate lambda terms using call-by-name/call-by-value strategies?
- What is the typing relation?
- What is type safety?
- What is erasure of types?
- What is general recursion?

Typical questions for seminar classes (labs) within this section

- Evaluate a given lambda expression using call-by-name strategy.
- Convert a given lambda expression to/from a nameless representation.
- Draw a type derivation tree for a given lambda term in simply typed lambda calculus.
- Provide a type for a given lambda term in a given simple type system.

Test questions for final assessment in this section

- Present an implementation of an interpreter for untyped lambda calculus.
- Present an implementation of a type checker for simply typed lambda calculus.

Section 2

Section title: References, exceptions, imperative objects, Featherweight Java

Topics covered in this section:

- References, store typings, raising and handling exceptions
- Subsumption and the subtyping relation, coercion semantics, the Bottom Type
- Object-oriented programming and lambda calculus with imperative objects
- Featherweight Java

What forms of evaluation were used to test students' performance in this section?

Form of evaluation	Usage
Development of individual parts of software product code	1
Homework and group projects	1
Midterm evaluation	1
Testing (written or computer based)	0
Reports	1
Essays	0
Oral polls	1
Discussions	1

Typical questions for ongoing performance evaluation within this section

- How operational semantic changes when introducing references?
- How operational semantic changes when introducing exceptions?
- What is the concept of imperative objects?
- What are the features of Featherweight Java?
- Explain effects of call-by-name and call-by-value evaluation strategies on terms with references.

Typical questions for seminar classes (labs) within this section

- Evaluate given expression with references.
- Evaluate given expression with exceptions.
- Draw a type derivation tree for a given lambda term in simply typed lambda calculus with references and exceptions.
- Provide a type for a given lambda term in a given simple type system with references and exceptions.

Test questions for final assessment in this section

- Present and/or explain an implementation of an interpreter for lambda calculus with references and exceptions.
- Present and/or explain an implementation of a type checker for simply typed lambda calculus with references and exceptions.

Section 3

Section title: Recursive types, type reconstruction, universal polymorphism

Topics covered in this section:

- Recursive types, induction and coinduction, finite and infinite types
- Polymorphism, type reconstruction, universal types
- System F and Hindley-Milner type system

What forms of evaluation were used to test students' performance in this section?

Form of evaluation	Usage
Development of individual parts of software product code	1
Homework and group projects	1
Midterm evaluation	0
Testing (written or computer based)	0
Reports	1
Essays	0
Oral polls	1

Form of evaluation	Usage
Discussions	1

Typical questions for ongoing performance evaluation within this section

- What is the concept of recursive types?
- What is the motivation for universal types?
- Explain the differences between System F and Hindley-Milner type system.

Typical questions for seminar classes (labs) within this section

- Evaluate given expression in System F with recursive types.
- Draw a type derivation tree for a given term in System F.
- Provide a type for a given term in System F.

Test questions for final assessment in this section

- Present and/or explain an implementation of an interpreter for lambda calculus with references and exceptions.
- Present and/or explain an implementation of a type checker for simply typed lambda calculus with references and exceptions.
- Present and/or explain the Hindley-Milner type inference algorithm.

Section 4

Section title: Compiling lazy functional languages

Topics covered in this section:

- Challenges of compiling lazy functional languages
- Representing functional closures at run-time
- Representing lazy data structures at run-time
- The syntax and semantics of the STG language

What forms of evaluation were used to test students' performance in this section?

Form of evaluation	Usage
Development of individual parts of software product code	1
Homework and group projects	1
Midterm evaluation	0
Testing (written or computer based)	0
Reports	1
Essays	0

Form of evaluation	Usage
Oral polls	1
Discussions	1

Typical questions for ongoing performance evaluation within this section

- How are closures represented during run-time?
- How are lazy data structures represented during run-time?
- Describe the main constructions of the STG Language?

Typical questions for seminar classes (labs) within this section

- Explain how a given term in STG language evaluates.
- Translate a given lambda term into STG language.

Test questions for final assessment in this section

- Present and/or explain the STG machine.