# Lab 5
# Projects

15-417/817: HOT Compilation
Frank Pfenning

Due Fri Mar 25
250 points

In this fifth lab you will explore a topic of your own choosing. We strongly suggest that you consult with the course staff after picking a topic, especially if you design your own project.

## 1 Submissions

Your submissions should be handed in directly to Gradescope. There will be two parts to hand in: code and data (such as benchmarks) and a write-up (term paper). The code and data will make up 150 points of your score, the write-up 100 points. So you should make sure to allocate sufficient time to present your findings.

### 1.1 Code and Data

Your code and data should have: A `readme.txt` or `readme.md` file that explains software dependencies and instructions to build your implementation, how to run the examples or benchmarks, and how to interpret the data. If you extend the ND or Sax grammars, it would be helpful to have a specification of the grammar.

It should also contain a guide to the examples or benchmarks with a short description what each of them does.

### 1.2 Term Paper

The term paper should have at least the following components. There are no page limits, although experience suggest that fewer than 5 pages are rarely sufficient to detail project outcomes. You may find the LaTeX sources for the lecture notes helpful; you can find them at the course resources page.

1. Title. The title of your project, such as "*Effective checking of ordered types for functional programs*" or "*An evaluation of basic optimizations for compiling functional programs with adjoint types*".
2. Abstract. A brief (150-250 word) summary of the problem, your approach, and your findings.
3. Introduction. Lay out the problem addressed, outline the approach taken, and summarize the findings.

4. Technical realization. Explain the structure and key properties of the implementation, including any unexpected obstacles you encountered and decision you made. There should be enough information for the course staff to understand your implementation in the context of the course, so this section may have to contain some information on choices you made in Labs 1–4.

5. Evaluation. This important section should be a critical assessment of the outcome of your project, including goals achieved and goals not achieved. Depending on your project, that may include specific examples that can be expressed or not expressed, and/or graphs and tables summarizing experimental outcomes.

6. Related work [graduate numbers only]. Departmental policy is for the graduate version of the course to be somehow distinguished from the undergraduate version, so a section on related work and citations is required for the graduate version only. Everyone should feel free to provide it, though.

7. Conclusion. This should briefly reiterate the main findings (now that the technical part has been described) and point to future work as appropriate.

## 2 Some Suggested Projects

You may choose your own topic that is related to the course material. Here are a few suggested topics that we deem to be likely feasible and rewarding, in no particular order. Several of them suggest measurements characteristics at the level of an interpreter (like the number of steps taken, or the number of memory cells allocated), in others you can measure the runtime and space of compiled code.

**Compile ND via Sax to a lower-level language and from there to a binary executable.** Possible choices for the low-level language are C (used by the research compiler), Go, Rust, or LLVM. In some cases, this would have to involve closure conversion. For this option, fixed width integers should be supported so some more realistic code can be written in ND and its efficiency after compilation assessed to identify some strengths and weaknesses of the compiler.

**Ordered types.** Build a type-checker for ordered types, including also at least linear and structural types, and extend the compiler to Sax. Taking this further could consider if there are any ways to exploit order in the dynamics of the Sax interpreter.

**Snax and data layout.** Retarget the compiler from Sax to Snax in order to integrate data layout. In the evaluation, some quantitative comparison with Sax might be interesting, for example, on memory usage or data locality.

**Garbage collection.** Implement one or more garbage collectors in the Sax interpreter. Among the choices are a copying collector, typical for functional languages, and a reference counting collector. Some quantitative measurements of the garbage collector(s) could be interesting.

**Parallelism.** Implement an interpreter for Sax that simulates maximally parallel evaluation of Sax and computes work and span of various algorithms. Clearly, these abstract measurement are not realistic when compared to a parallel compiler such as CMU's MaPLe, but could nevertheless provide some insight. Can we somehow quantify the impact of substructural types in this context?

**Optimizations.** We have considered various optimizations that could be implemented and their efficiency assessed abstractly, at the level of a sequential interpreter. This includes cut/id optimizations, silent implementations of unit, or vertical and horizontal reuse. It could also include other optimizations such as tail call optimizations, or optimizing closure conversion.