I'm on vacation in Italy and typing on an old iPad with horrible internet, so I literally do not have the technological capability to submit a pdf. So, I'm just entering this as a comment (attached is a dummy file that I generated and downloaded from the internet, I genuinely can't find an iOS way to turn a doc into a pdf).

I do not have a partner and will miss tomorrow's lecture, so I'll do my best to figure something out. I really want to do something like my project idea (or at least, related to sub typing) but worst case, I'll abandon my idea and work on something else. What I have here is something that I'd love to turn into a blog post. The reason I wanted to take this class is to learn language features from first principles (which is what type theory is all about, as I understand it) and unlearn mere language implementations of those features, so if we can go in that direction, I'd be very grateful.

Anyway, here's what I'm proposing;

Getting Our Ducks In A Row Or, Implementing Type Checkers To Investigate Duck Typing

Motivation

A structural type system is one where a term's type is determined by its structure rather than anything else (commonly, the name of its declared type). There appears to be some confusion about the distinction between structural subtyping and duck typing, especially of the implementations of these techniques in languages such as Go and Java (or at least, particular features of Java). This project will explore that distinction in cases of subtyping by implementing a simple type checker, and will also discuss several implementations of duck typing in commonly-used languages as well as their features similar to duck typing

Proposal

I will implement a few small type checkers which will each, given two well-typed expressions, output whether those expressions are equivalent. The first will be a type checker for the dynamic simply-typed lambda calculus (Henglein, 1992) which uses structural subtyping. The next two will attempt to demonstrate duck typing (i.e. perform a check on a more limited set of a value's structure, given some task at hand), so I will attempt to implement one type checker in the static case, and one in the dynamic case. The goal will be to use these implementations as a basis for comparing structural subtyping with duck typing.

Background and Related Work

Henglein extended the simply-typed lambda calculus to include the notion of dynamic typing by introducing coercion. In his paper, Henglein works with functions and also introduces booleans in order to allow for the possibility of type errors in his system (since if it were only functions, there would be no type errors). In order to make the scope of my project feasible, I plan to follow the

example in this paper by limiting the values that I will work with to just the primitives of the lambda calculus, plus booleans.

Sources:

"Dynamic Typing: Syntax and Proof Theory", F. Henglein. Science of Computer Programming 22 (1994). "Is Go Duck-Typed?", Ian Smith. https://www.fullstory.com/blog/is-go-duck-typed/ and Hacker News discussion: https://news.ycombinator.com/item?id=22486470