Progress Report Alpha

Evan Hansen

November 13, 2020

Vision: The vision of the project is to build a small functional language based on the lambda calculus, and ideally implementing as many real-world features that appear in languages like ML as possible. After the lecture on type inference, I think an achievable goal would be to implement ML style type inference and polymorphism. I also plan on implementing recursive types and the ability to make new type constructors, but this will depend on how much time I have. If I succeed in doing all those things, I thought it would be cool to try to support functors or monads with nice syntax. However, I have not had much success in understanding how to do lexing and parsing, so may have to decide whether to spend time implementing more features in the language or making the language actually usable, and am not sure which I will do.

Summary of Progress: In the beginning of this project, I wanted to do type-inference but did not really look into how it was done, and thought I could just figure it out. That didn't really work out so I decided to rebuild the system and currently have something like the simply-typed, applied lambda calculus extended with let statements and fix point operators (the syntax is going to be let rec). Currently, there are basically two levels to the language: a base level that is the untyped, applied lambda calculus, and a higher level that has let statements and let recs. The idea is that you can easily do the type checking at the higher level, and then when you translate let recs or other constructs into the base lambda calculus language, you don't have to worry about whether your translations of these things can be well-typed; for instance right now let recs are implemented with the Z combinator even though my type system cannot type the Z combinator. I haven't had much time to work on the project recently, but think that over the break I will redesign the type system and type inference, ultimately getting to a system that uses unification to do ml-style type inference. One concern I have is that I am not sure how much of the code I wrote for type checking in this sprint will be usable when I implement type inference.

Activity Breakdown: I am working alone, but see 'summary of progress' for what I have done so far.

Productivity Analysis: I was fairly productive and reached the goals that I had set out, but a change in my knowledge of type systems has made me think that implementing the type checker for the simply-typed calculus may end up being wasted effort when I implement type inference; however, I am also more confident in my knowledge so think it will not be too difficult to implement the new type checker, and the syntax of the language probably will not need to be changed.

Grade: Good

Goals:

Excellent: implement ML-style type inference and polymorphism, recursive types, and lexing and parsing

Good: either implement ML-style type inference, polymorphism, and recursive types or build a lexer and parser

Satisfactory: implement type-inference

How to Use: So far, I have been testing my implementation by keeping tests that I want to run in tests.ml and then running ocamlbuild tests.native && ./tests.native to build and run the tests, which print out debugging info.