Lambda Calculus with Lifetimes and Higher Kinded Types

Final Year Project Report

Felix Bowman
Candidate Number: 122587

Supervisor: Dr. Martin Berger

University of Sussex
Department of Informatics

Contents

Listings

Chapter 1

Introduction

1.1 Project Aims

This project aims to design and implement a language with a novel type system, with two main aspects. The first is the ability to abstract over type constructors as well as types, in effect allowing for a language at the level of types as well as expressions. The second is a memory management technique that guarantees memory safety at compile time. This technique is incorporated into the language as part of the type system. These two concepts will be implemented in the Lambda Calculus. A type checker and interpreter will be developed with Higher Kinded Types and a region-based memory management model similar to that of the Rust programming language.

The end purpose of this project is to investigate how Higher Kinded Types management with the existing type system of the Rust programming language, which already has a system of guaranteeing memory safety.

1.2 Objective: Combining Type Systems

The concepts described in Sections **??** and **??** are combined in the Lambda Calculus to study how they interact and uncover problems that may arise. The syntax of this language is outlined in Appendix **??**.

1.2.1 Base Objectives

The base objectives for the project are outlined below. These are objectives that should be reasonably achievable in the time given.

- Design a language which incorporates region based memory management techniques and Higher Kinded Types into the type system. Describe the language using a formal grammar.
- Implement the Lambda Calculus extended with references as a Haskell program, modelling dynamic memory allocation inside the interpreter for the language.
- Implement a system for ensuring all resources in the language have exactly one owner (are assigned to one variable), based on the model in Rust.
- Implement a type system that incorporates higher order polymorphism into the language (higher kinded types).
- Formalize the rules of the type checker and construct the appropriate typing derivations.

1.2.2 Extensions

Some extensions to the project are outlined below, which should be completed depending on time and complexity constraints.

- Extend the base lambda calculus with constructs that more closely model the Rust programming language, including traits (or in the case of the langauge outlined in the project, type constructor classes), enums (discriminated union types), and local type inference.
- Investigate how concepts learned in this project can be incorporated into `rustc`, specifically adding Higher Kinded Types to the language.

1.3 Project Overview

The rest of this report is divided in several chapters. Chapter 5 describes concepts neccessary in order to understand the rest of the report. Chapter 2 gives an analysis of the ethical considerations of this project. Chapter 3 list the requirements of this project. This chapter is divided into the functional requirements, or what the project will do, and non-functional requirements which list how the project will be carried out. Finally the acceptance criteria are listed, which the final tests of the project will be based on. Chapter **??** breaks down the main phases of work to be completed in this project and gives an estimated time for each phase. Work done so far is also listed. Chapter **??** details what is discussed during meetings. Both meetings which have already happened and meetings which have yet to happen are listed. Appendix **??** gives the original project proposal which was already submitted. Appendix **??** describes the grammar of the extended Lambda Calculus that this project is based on.

Chapter 2

Professional and Ethical Considerations

No part of this project requires human participation and as such there are no ethical considerations.

This needs more detail, perhaps licensing issues?

Chapter 3

Requirements

The functional requirements of this project specify what this project shall do.

3.1 Functional Requirements

The functional requirements of this project are laid out in this section. Because the entire system is a Haskell program all of these requirements will be implemented in Haskell.

3.1.1 Parser

Description	A parser for the language specified in **??** shall be created using the `megaparsec`[1] parser combinator library.
Input	Source code from a file or interactive session.
Output	A data type that represents the abstract syntax of the input provided, or an error message pointing to the location of any syntax errors.
Error	An error message with a line and column number and a message.

3.1.2 Simple Type Checker

Description	A simple type checker that ensures that simple mistakes are not make, e.g. using a function type where a numerical type is expected.
Input	A syntactically valid (according to Figure **??**) AST of a program as a Haskell data type.
Output	The same AST of the program that confirms the typing rules of the language.
Error	An error with a message that provides some indication of what went wrong.

3.1.3 Ownership Checker

Description An ownership checker ensures there exactly one binding to a resource all the time.

Input A syntactically valid (according to figure **??**) AST of a program as a Haskell data type.

Output The same AST of the program that confirms the ownership rules of the language.

Error An error with a message that provides some indication of what went wrong.

3.1.4 Borrow and Lifetime Checker

Description A checker that ensures that references that borrow ownership from another type last longer than the resource they borrow, and that there are only mutable references OR exactly one mutable reference at any one point.

Input A syntactically valid (according to figure **??**) AST of a program as a Haskell data type.

Output The same AST of the program that confirms the borrowing rules of the language.

Error An error with a message that provides some indication of what went wrong.

3.1.5 Kind Checker

Description Ensures that all type constructors using in a program have the correct number and kind of arguments.

Input A syntactically valid (according to figure **??**) AST of a program as a Haskell data type.

Output The same AST of the program that confirms the kinding rules of the language.

Error An error with a message that provides some indication of what went wrong.

3.1.6 Evaluator

Description A call-by-value evaluator of the language that will reduce a syntactically valid expression.

Input A syntactically valid and type-checked AST of a program represented as a Haskell data type.

Output The final resulting value of evaluating the AST.

Error A description of any runtime errors that occur within the program.

3.1.7 Interactive Interpreter

Description	An interactive interpreter that type checks and then evaluates entered expressions.
Input	Source code as entered by the user.
Output	The resulting value of evaluating the entered expression, some error.
Error	An parsing, type, or runtime error message.

3.1.8 Load a file

Description	Provided with a path, the program loads a text file containing source code.
Input	A path provided by the user.
Output	The source code as a string.
Error	An error reporting a file not found or any other errors.

3.2 Acceptance Criteria and Testing

The acceptance criteria of this program correspond to the functional requirements in Section 3.1. The finished project should pass the tests laid out in this section.

3.2.1 Parsing

Functional Requirement	3.1.1
Passing Criteria	The program should be able to parse valid source code and correctly report any errors that are encountered.
Tests	Test numbers

3.2.2 Type checking

Functional Requirement	3.1.2, 3.1.3, 3.1.4, 3.1.5
Passing Criteria	The type checker should detect any errors in the program.
Tests	Test numbers

3.2.3 Evaluating

Functional Requirement	3.1.6, 3.1.7, 3.1.8
Passing Criteria	Source code, provided by a file or through the interactive interpreter, should be type checked and evaluated.
Tests	Test numbers

Chapter 4

Background

Chapter 5

Motivation

5.1 Higher Kinded Types

5.1.1 Parametric Polymorphism

generics are found in languages like java
reducing code duplication

list example

The notion of first-order generics in programming langauges have been formalised System F [2].

5.2 Lifetimes

Chapter 6

Implementation of `lambda-calc`

Chapter 7

Testing

Chapter 8

Evaluation

Chapter 9

Conclusion

Bibliography

[1] *megaparsec: Monadic parser combinators*. `https://hackage.haskell.org/package/megaparsec`. Accessed: 2016-10-18.

[2] Adriaan Moors, Frank Piessens, and Martin Odersky. "Generics of a Higher Kind". In: *SIGPLAN Not.* 43.10 (Oct. 2008), pp. 423–438. ISSN: 0362-1340. DOI: `10.1145/1449955.1449798`. URL: `http://doi.acm.org/10.1145/1449955.1449798`.