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

4.1 Lambda Calculus

4.2 System F

4.3 First Order Polymorphism

Chapter 5

Motivation

5.1 Higher Kinded Types

5.1.1 Parametric Polymorphism

First order parametric polymorphism, know as Generics in Java, are types that are parametrized over some other type. They used to write code that can be checked for safety at compile time. For example, without generics, a list could be used like:

```
List l = new ArrayList();
l.add("This␣is␣a␣string");
Integer i = (Integer) l.get(0); // Run time error here
```

Listing 5.1: Runtime error that could be avoided

This is problematic because a list in Java can contain any type of object, but methods that the `List` object provide must know about the type of object that the list contains. If those methods do not have a way of knowing what kind of object a list contains then calling the methods will not be type safe, as demonstrated in Listing 5.1. The same code written with Java's Generics will produce a compile time error:

```
List<String> l = new ArrayList<String>(); // Now the list has been parametrized with a type
l.add("This␣is␣a␣string");
Integer i = l.get(0);                     // Compile time error here.
```

Listing 5.2: Compile time error

Compile time errors are much more desirable then runtime errors because they can be caught and fixed at a predictable time, unlike runtime errors which may happen at unpredictable times. The code in Listing 5.2 parametrizes the `List` type with the `String` type, and hence the last line produces a compile time error as the list's `get` method returns a `String`.

Parametric polymorphism is an important addition to statically typed programming languages. However, Generics in Java and in lots of other programming languages have the limitation that types can only be parametrized with other types. This leads to some important limitations.

Talk about formalization, System F

5.1.2 Higher Order Polymorphism

As mentioned in Section 5.1.1, First order parametric polymorphism can be very useful in expressing certain concepts succinctly in programming languages. However, there are limitations. This section will attempt to demonstrate one shortfall of first-order polymorphism and then show how the problems can be solved with Higher Order Polymorphism.

Functors

```
interface Functor<A> {
    Functor<B> map(Function<A, B> f);
}
```

Listing 5.3: An attempt to define Functor in Java

The code in Listing 5.3 has one main issue. The `map` method defined here may return any type that implements `Functor`, not the necessarily the same class as the one that the method has been called from. This means that there is no type-safe way of calling any method on the result of calling the `map` function.

In order to define a generic Functor interface, a way of referencing the type constructor that is being used as a functor is needed. In other worlds, the programming language needs type constructors that can be parametrized with other type constructors. Generics in Java provide a way of making types like `Integer`, `String`, and even `List<Integer>` first class, but type constructors like `List` must be applied to some concrete type before they can be abstracted over.

```
class Functor f where
    fmap        :: (a -> b) -> f a -> f b
```

Listing 5.4: Functor as defined in Haskell.

Listing 5.4 shows how Functor can be defined in a Haskell, a language that does allow higher order polymorphism. Here, `f` is a variable that references a type constructor. This class in Haskell specifies that anything that instanciates it must provide a single function, `fmap`. `fmap` takes a function as a parameter that takes a value of type, `a`, and returns a value of type `b`. It then takes another argument of type `f a`, or a type that the type constructor `f` has been applied to. For example, it would take a value of type `Maybe Int`, `Maybe` being the type constructor `f`. It would then return a value of type `b`, wrapped in the same type constructor `Maybe`.

The ability to use type constructor as first class in a programming language allows programs to be written in a more succinct, patterned, and generic manner. Examples like Functor show how a generic interface to work with constructs commonly found in software engineering can be achieved with the use of Higher Order Polymorphism.

This problem can be solved with HKTS...

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

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.