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 4.3.1 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 3 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 6 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

Background

This chapter introduces some technical terms, technologies, and systems that will occur throughout the rest of this report.

3.1 Haskell

Chapter 4

Motivation

CHAPTER INTRO HERE

4.1 Higher Kinded Types

This chapter introduces Higher Kinded Types, also known as higher order polymorphism, and show what kinds of problems a language that includes these features can solve.

4.1.1 Parametric Polymorphism

First order parametric polymorphism, known as Generics in Java, are types that are parametrized over some other type. This kind of polymorphism allows for the definition and use of functions that behave uniformly over all types. They can be 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 4.1: 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 4.1. 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 4.2: Compile time error

Compile time errors are much more desirable then runtime errors because they can be caught and fixed predictably, unlike runtime errors which may happen at unpredictable times. The code in Listing 4.2 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

4.1.2 Higher Order Polymorphism

As mentioned in Section 4.1.1, 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

A Functor is important concept in modern programming languages. Most languages have constructs that can be thought of as Functors, such as lists, optional types, trees, and other constructs that can be mapped over. Anything that acts as a container for another type and provides a function for mapping a function over that contained type is a Functor. They are an important concept because they provide a common interface to working with any type that acts as a box for another type. This means that programs can be written more generically and is more open to changes and modifications.

Some languages seek to reify the general concept of functors so that aspects of a program can be checked for correctness at compile time. However, expressing Functors in a generic way requires a more expressive type system, specifically a type system that incorporates higher order polymorphism. Here is an example of an attempt to define a general Functor interface in Java:

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

Listing 4.3: An attempt to define Functor in Java

The code in Listing 4.3 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 4.4: Functor as defined in Haskell.

Listing 4.4 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 constructors 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.

Kinding

Higher order polymorphism can be thought of as the polymorphic lambda calculus 'one level up', and as such the language at the level of types also needs to have a type system in order to prevent wrong expressions from being created.

A system that allows for abstracting over type constructors needs some way of enforcing type constructors are not applied to types in a way that does not make sense, e.g. `Integer Integer` does not make sense because the integer types does not take any type arguments in order to become a concrete type that can be inhabited by values. Alternately

just `List` cannot be instantiated with a value because it need one more type constructor, the type that will be contained within that list.

What is needed is essentially a type system for types. This is the notion of the kind of a type. Types can have the kind of $*$, in which case they are concrete and can be inhabited by values, or they can have kind $* \to *$, or a function from types to other types. Types like `Integer` or `List Integer` have kind $*$ however type constructors like `List` have kind $* \to *$ because they need to take one more concrete types before they can be used as a value. This system of kinding enforces that all types are well formed.

$$\begin{array}{rcll} K & ::= & * & \text{Concrete types} \\ & | & K \to K & \text{Type functions from types to types} \end{array}$$

Figure 4.1: Syntax of types

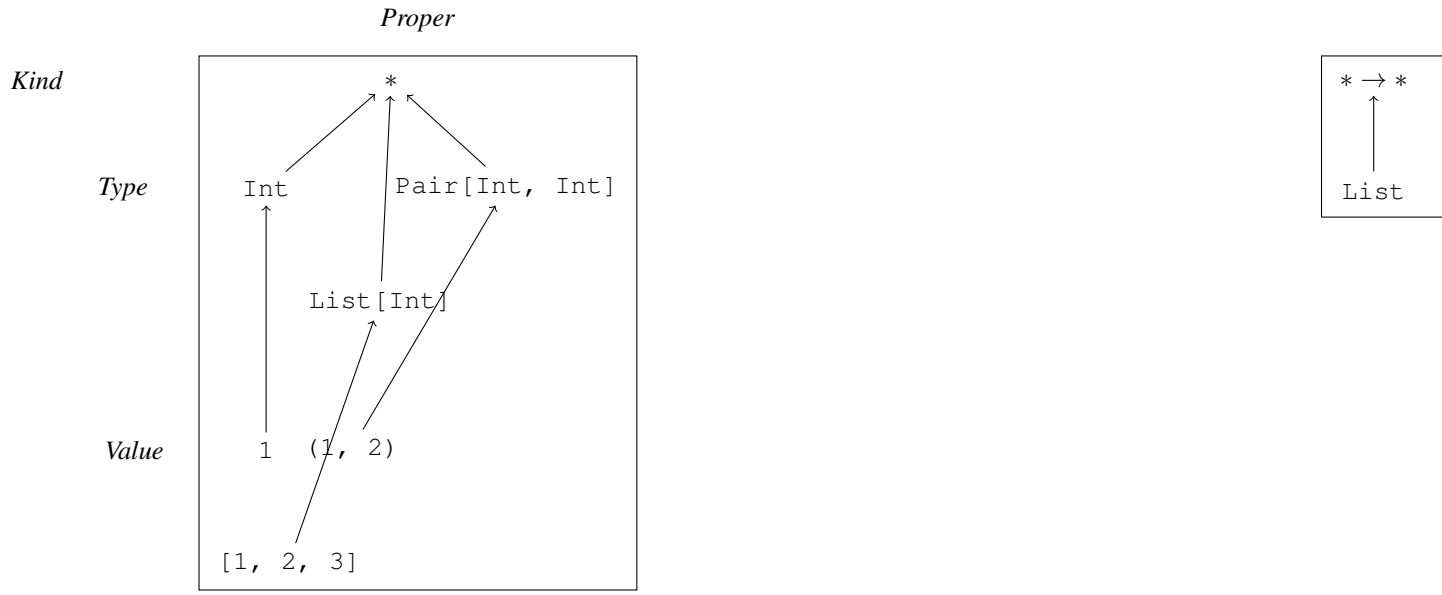

Figure 4.2: Some types and their associated kinds.

4.2 Problems with Existing Memory Management Techniques

Most programming languages allow for dynamic memory management where the program can request portions of memory at run time and free that memory when it is no longer required. This operation can be performed manually as in the case of languages such as C or automatically in languages with a garbage collector such as Java.

4.2.1 Manual Memory Management: `malloc` and `free`

Some languages rely on manual instructions inserted by the programmer in order to allocate and deallocate regions of memory. Most implementations of the C programming language provide a group of library functions for this purpose, which include `malloc` and `free`. As mentioned, these constructs are insert manually by the programmer. This can lead to the issues described below.

Dangling References

Deallocating storage may lead to dangling references, where some reference exists that points to memory that has been deallocated. This is nearly always unintentional, however it is an easy for a programmer to make this mistake when writing a program. Using the value of deallocated storage can lead to nefarious bugs and most languages consider using deallocated storage to be undefined behaviour.

```
int *dangle()
{
    int i = 69;
    return &i; // Function return address of local variable!
}

int main()
{
    int *p;
    p = dangle();
}
```

Listing 4.5: A C program that leaves `p` pointing to deallocated memory.

In the Rust programming language, references to values must live longer than the resource that they refer to [**rust-borrowing**]. The same program as the C dangling pointer example (Listing 4.5) written in Rust does not compile as it does not meet this restriction.

```
fn dangle() -> &u32 {
    let i = 69;
    &i // This does not typecheck because this would be a dangling pointer
}

fn main() {
    let p = dangle();
}
```

Listing 4.6: A Rust program that does not type check.

Memory Leaks

Memory leaks occur when heap storage that is no longer usable is not deallocated. This can lead to software allocating more memory for itself and in the worst case can lead to a program consuming all available memory.

```
#include <stdlib.h>

void allocate()
{
    int* a = malloc(sizeof(int) * 100); // Allocate memory on the heap
    // Return and forget about the allocated memory!
}

int main()
{
    allocate();
    // ...
    // do more stuff, while heap space that allocate used is unreachable
}
```

Listing 4.7: A C program which leaks memory.

4.2.2 Automatic Memory Management: Garbage Collection

Techniques have been invented in order to mitigate errors related to manual memory management. Garbage collection is one such method. Garbage is a term for storage that has been allocated on the stack but is no longer reachable by the

program, e.g. the storage is pointed to by a variable that has gone out of scope. Automatic garbage collection alleviates the programmer of explicitly managing memory. However there are several downsides to garbage collection. For example, the garbage collector will be invoked to collect unusable memory outside of the control of the programmer. When it is running it affects the execution of the program for an indeterminate amount of time. This hang is not acceptable in real–time systems.

4.3 A Solution in the Type System

The Rust programming language achieves memory safety through its ownership rules and borrowing system [**rust-borrowing**]. The ownership system allows Rust to achieve this memory safety by enforcing typing system rules at compile time. This means that there is no run time overhead in a Rust program as there would otherwise be in a garbage collected language.

4.3.1 Ownership, Borrowing, and Lifetimes

In Rust, a resource must have one owner, a stack variable. When a variable goes out of scope, the resource that it owns is automatically freed. There cannot be more than one variable pointing to a given resource at any time. The reason for the "use of a moved value" error often encountered in code written by novice Rust users is because of this rule. This prevents any two parts of a Rust program from accessing a section of heap memory at the same time.

Rather than passing ownership of a resource around between functions, Rust allows for a reference to be borrowed by another scope temporarily. When references go out of scope the resource that they point to (have borrowed) do not get automatically freed. This means that after a function with a reference passed in as as parameter returns the resource can be used again. The scope of borrow must always be shorter than that of the resource which it borrows. This concept is formalized in Rust as Lifetimes. The type system will enforce that a borrow is not made of a resource that will be freed before the borrow ends.

There are two kinds of references: immutable and mutable. There many be many immutable borrows of a resource at once or a single mutable borrow. This also prevents heap data from being accessed at the same time at different points in the program.

This system where resources are freed after the single owner goes out of scope, as well as borrows always having shorter lifetimes than original resource mean that Rust can enforce the memory safety of programs. These concepts are built into the type system.

Chapter 5

Formal Definitions

As described in Section 3.1, the lambda calculus can be used to formally reason about aspects of programming languages. In this way the lambda calculus is a very good starting point when looking at how new language features should behave.

An attempt is given in this section to formally specify aspects of the language implemented in this paper.

5.1 Lambda Calculus

The lambda calculus is model of computation where the only behaviour is function definition and application. It is commonly used to specify programming language features and to formalize their behaviour. Specifically, it is commonly used to reason about type systems. The lambda calculus can be viewed as a miniature programming language and a system where strict properties can be proved.

t	::=	x	variables
	|	λx . t	function abstraction
	|	t t	function application

Figure 5.1: Grammer of the untyped lambda calculus.

5.1.1 Type in the Lambda Calculus

The type system introduced in this paper introduces extensions to the simply typed lambda calculus. The higher order polymorphic lambda calculus, known as System F ω, is included as well as a lifetime–calculus. Both are described in Chapter 5

5.2 System F

System F is an extension of the simply typed lambda calculus [**tapl**] that allows for quantification over types as well as terms. In doing this it formalizes the notion of polymorphism in programming languages, as described in Section 4.1.1. It is used to study implementations of polymorphism in programming languages.

As the simply typed lambda calculus allows abstraction of terms outside of terms through function definitions, System F introduces abstractions at the level of types. The system also allows for application of type level expressions.

`id` = $\lambda X.\lambda x : X.x$

Listing 5.1: Polymorphic identity function showing type level abstraction.

The example in Listing 5.1 shows type level lambdas, where X is a type variable that has been abstracted out of the function definition. The term level lambda abstracts x which has the type X. Applying `id` to a type maybe be written as `id [Int]`, which would yeild

5.3 System F ω

System F ω is form

5.3.1 Type Operators

5.4 Adding Lifetimes

showing how lifetimes are represented in this lanaguage

5.5 Typing Rules

Chapter 6

Requirements

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

6.1 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.

6.1.1 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.

6.1.2 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.

6.1.3 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.

6.1.4 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.

6.1.5 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.

6.1.6 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.

6.1.7 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.

6.1.8 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.

6.2 Acceptance Criteria and Testing

The acceptance criteria of this program correspond to the functional requirements in Section 6.1. The finished project should pass the tests laid out in this section.

6.2.1 Parsing

Functional Requirement	6.1.1
Passing Criteria	The program should be able to parse valid source code and correctly report any errors that are encountered.
Tests	Test numbers

6.2.2 Type checking

Functional Requirement	6.1.2, 6.1.3, 6.1.4, 6.1.5
Passing Criteria	The type checker should detect any errors in the program.
Tests	Test numbers

6.2.3 Evaluating

Functional Requirement	6.1.6, 6.1.7, 6.1.8
Passing Criteria	Source code, provided by a file or through the interactive interpreter, should be type checked and evaluated.
Tests	Test numbers

Chapter 7

Implementation

The language described in Chapter 5 has been implemented as a Haskell program. Haskell was a natural choice as it is well suited to building type checkers and interpreters. It also has the benefit of having higher kinded types built into the language, making it ideal for testing potential features of the implemented language.

7.1 Parsing

A lexer generator Alex [**alex**], and parser generator, Happy [**happy**] were used in this project. Using these tools in combination made parsing the language into the Haskell representation of the abstract syntax simple. The grammars provided for the parser generator are a very close approximation of Haskell data type. Both Alex and Happy generate Haskell files when run which implement the specified grammars.

7.2 Terms

The Haskell representation of terms of the language are given in Listing 7.1. Type lambdas and Lifetime lambdas can be seen as first class citizens here.

```
data Term
    = Var String
    | Lit Int
    | Lam String (Lifetime, Type) Term
    | App Term Term
    | TyLam String Kind Term
    | TyApp Term Type
    | LiLam String Term
    | LiApp Term Term
    | Lt Lifetime
    deriving (Eq, Show)
```

Listing 7.1: Haskell representation of Terms.

7.3 Kind Checker

The kind checker makes sure that the type expressions in a supplied program are well–formed. The Haskell abstract data type that represents kind expression is show in Listing 7.2.

```
data Kind
    = KnStar
    | KnArr Kind Kind
```

```
    deriving (Eq, Show)
```

Listing 7.2: Haskell representation of Kinds.

7.4 Lifetime Checking

The syntax of lifetime is show in Listing 7.3.

explain what each of the lifetime consturctors mean

```
data Lifetime
    = LiVar String
    | LiLit Int
    | LiStatic
    | LiDummy
    deriving (Eq, Show)
```

Listing 7.3: Haskell representation of Lifetimes..

`LiDummy` is used here as a dummy place holder for occurrences of lifetime literals during parsing, as lifetimes are associated with the scope of terms. An initial walk of the tree replaces dummy values with the `LiLit` value that represents the depth of the scope that the lifetime value is found in.

7.4.1 Ordering on Lifetimes

should probably write this, Ord instance for lifetimes.

7.5 Type Checking

Type checking is rather involved in this this language as the type system almost contains the lambda calculus itself.

```
data Type
    = TyVar String
    | TyInt
    | TyArr Type Type
    | Forall String Kind Type
    | OpLam String Kind Type
    | OpApp Type Type
    deriving (Eq, Show)
```

Listing 7.4: Haskell representation of Types.

7.6 Error Reporting

Any part of the program that may result in some kind of error is wrapped in a partial application of Haskell's error monad, `Except`, to a custom error data type:

```
type ThrowsError = Except LangErr

data LangErr
    = ParseError
    | NoMain
    | VarNotFound String
    | WrongKind Kind Kind
    | NotKnArr Kind
    | WrongType Type Type
    | NotTyArr Type
    | NotForall Type
    deriving (Eq, Show)
```

Listing 7.5: Partially applied error monad and langauge errors.

This has the advantage of being very composeable, and also of reducing error handling boilerplate in the program.

7.7 Context Management

The lifetime checker, kind checker, and type checker all rely on a variable typing context. These contexts are threaded thought the program using Haskell's environment monad, also known as the Reader monad. The record type holding these contexts is shown in Listing 7.6. Contexts are represented as a map from variable names are strings to some value, using Haskell's own built in strict map data structure.

```
type Ctx = Map.Map String

data Env = Env
    { _typeCtx :: Ctx (Type, Lifetime)
    , _kindCtx :: Ctx Kind
    , _ltCtx :: Ctx Lifetime
    } deriving (Show, Eq)
```

Listing 7.6: Record data type showing contexts.

The monad stack where parsing, lifetime checking, kind checking, and type checking take place is therefore:

```
type Typing = ReaderT Env ThrowsError
```

Listing 7.7: Envirnment and error monad stack.

7.8 Testing

Chapter 8

Key Tests

Chapter 9

Evaluation

Chapter 10

Conclusion

Bibliography

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