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

The purpose of this project is to design a type system for a programming language that incorporates Higher Kinded Types with a system that guarantees memory safety at compile time. Higher Kinded types are necessary to neatly express certain desirable programming features. Rust is a programming language that provides memory safety through compile-time checks. This aspect of Rust is known as the *ownership* model, of which *lifetimes* and *borrowing* are key concepts [**rust**]. This project attempts to implement a type checker for this language as a Haskell program, and investigate how higher kinded types and the Rust ownership model interact.

1.2 Types in Programming Languages

Type systems are designed to help programmers reason about behaviours in the programs that they write. Types restrict the kinds of statements that may be expressed, the most obvious reason for doing this is to detect certain types of errors before a program is executed. Small, obvious errors, found in expressions such as `"this sentence" * 69` are easily detected, because multiplication makes no sense when applied to a strings of characters and a number. However, modern type systems are capable of doing far more.

As programs become larger and more complex, separating parts of programs that do not need any knowledge of each other becomes more important. It is much easier to maintain a program composed of separate modules that have no knowledge of the inner workings of one another. Modules may interact through well defined interfaces and in this way may be swapped out or replaced much more easily. One may re-write code that conforms to an existing interface without knowing about another module's implementation details. A type system can be used to define this interface and to ensure that new code behaves as expected. Polymorphism, discussed in detail in Section 3.1.1, is one way of allowing more flexibility in these interfaces between parts of a program while retaining correctness.

Some programming language features may introduce whole new families of errors. Allowing a programmer to manually allocate and deallocate space for storing objects in memory is known to be a common source of bugs. Languages like C have a reputation for being difficult to write bug free programs in because of manual memory management. Memory leaks may occur if a programmer requests some storage in memory but does not deallocate it. This can lead to programs consuming unreasonably large amounts of memory. Referencing some location in memory after it has been deallocated (and possibly allocated with some new object) is another common pitfall associated with manual memory management. Garbage collection, or the automatic allocation and deallocation of objects in heap memory, may alleviate some of these issues. However, it can incur a sometimes unacceptable runtime overhead especially in software systems where responsiveness is key.

A solution can be found in the type system. Restricting where a programmer can request or reference some location in memory allows a program to be statically analysed before it is run, and any sections of code that could cause memory related issues are shown as compile time errors. A type system can form the basis of this restriction, as shown in Section 3.3.1

In studying these type systems it is often useful formalize them in a system with well known and well understood behaviour. In this case the system is the lambda calculus, which functions as a miniature programming language and as a formal model for understanding the behaviour of types. The lambda calculus and how it pertains to this project is described in Section 4.1. Higher Kinded Types, or higher order polymorphism, have been well described in System F ω, itself a typed lambda calculus [**tapl**].

1.3 Goals of This Project

A type checker for a language based upon the lambda calculus will be developed as a Haskell program. Haskell is described in Section **??** along with the reasons for choosing it. A parser, type checker, and simple interpreter will all be implemented along with a test programs demonstrating some intended behaviours. The design of the language is described in Section 4. The type-checker aspect of the program is concerned with:

- Higher Kinded types, described in Section 3.1.1. Formalised in System F ω, they allow for a neat implementation of very generic, high level programming concepts such as Monads and Functors.
- A system restricting the allocation, use, and deallocation of memory, encoded into the type system. This aspect is based on the type system of the Rust programming language, which guarantees memory safety through its lifetime and borrow checker.

The notion of an addressable area of memory, as well as mechanisms for allocating and deallocating areas of that memory, will be added to the base lambda calculus. The implementation of the type system, as well as the parser, build system, test framework, and interpreter are described in Chapter 6. A formal description of the language can be found in Chapter 4.

1.4 Project Overview

The rest of this report is divided in several chapters. Chapter 2 lists the ethical considerations of this project. Chapter 3 describes important concepts in more detail as well as describing how common tasks related to type checking are accomplished. Chapter 4 then introduces the lambda calculus as way of reasoning about type systems. The design of the language is shown including the grammar. Chapter 5 formalizes what should be accomplished by the type checker to be implemented. Software engineering issues are tackled in Chapter 6, including the test framework and build system used. Key tests demonstrating desired behaviour are given in Chapter 7. Issues about the effectiveness of this type system are described in Chapter 8. Finally, unmet requirements and further work to be done is covered in Chapter 9.

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 and Motivation

This chapter attempts to give some background for the topics covered in this project. Aspects from existing type systems are introduced and explained. Later an introduction to how programming lanagauges are represented as trees is given, as well as a how these trees can be traversed in order to build a type checker. *fix this sentence*

3.1 Higher Kinded Types

This section introduces Higher Kinded Types, also known as higher order polymorphism, and demonstrates the expressive power of this concept though examples.

3.1.1 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 3.1: 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 provides 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 3.1. 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 3.2: 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 3.2 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`.

Type Constructors

A type constructor is a feature of programming languages that builds new types from old ones. In Java, an example would be the `List` type. The `List` type cannot be used to describe some value in Java; all values must be a `List` *of*

something, for example a `List<String>`. In other words, `List` is a type constructor that must be parametrized with some base type like `Integer`, `String`, or even `List<String>`. Once it has been applied to some base type, it becomes a new type; `List<Integer>`, `List<String>`, and `List<List<String>>` are all separate types.

Parametric polymorphism is an important addition to statically typed programming languages. First order polymorphism is modelled by a typed lambda calculus called System F, which is introduced in Section 4.1.2. However, Generics in Java and in lots of other programming languages leave something to be desired. There are higher level, general, generic programming concepts that cannot be expressed with first order polymorphism. Generally, these systems are constrained in that type constructors may only be parametrized by concrete types, and not other type constructors. Higher order polymorphism is needed in order to neatly express some programming concepts.

3.1.2 Higher Order Polymorphism

As mentioned in Section 3.1.1, first order parametric polymorphism can be very useful to programmers. 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 are introduced as an example of one of the concepts that cannot be expressed neatly with first 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 3.3: An attempt to define Functor in Java

The code in Listing 3.3 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 3.4: Functor as defined in Haskell.

Listing 3.4 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 system in place 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 3.1: Syntax of types

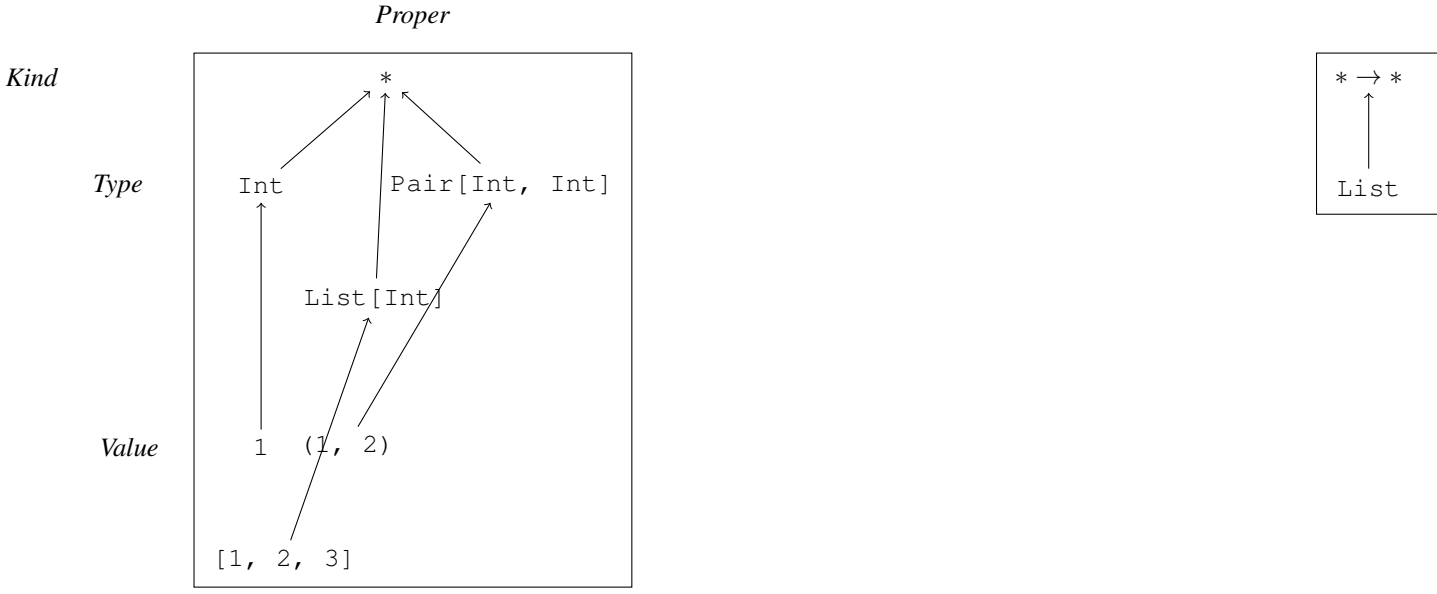

Figure 3.2: Some types and their associated kinds.

3.2 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.

3.2.1 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 3.5: 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 3.5) 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 3.6: 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 3.7: A C program which leaks memory.

3.2.2 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.

3.3 A Solution in the Type System

Memory safety can still be achieved without the use of a garbage collectors. Program structure may be analysed during semantic checking and areas likely to cause issue can be identified. Here Rust's way of performing these checks is examined.

3.3.1 An Introduction to Rust's model of Memory Management

Rust, a systems programming language [**rust**] is designed to provide memory safety with no runtime overhead. Rather than relying on a garbage collector to provide run-time memory safety, Rust relies on strong compile time checks. This system of compile time checks is referred to as *ownership*, and is derived from affine type systems and unique pointers to memory locations[**rust-borrowing**] [**levy2015ownership**].

Affine types

Affine types are a development of Linear type systems. Linear type systems ensure that every variable is used exactly once. In an affine type system, a variable may be used at most once, i.e. zero or one times. This has applications for managing resources in a program that do not persist the whole time that the program is running, e.g. file handles or heap allocated memory [**attapl**] [**tovAffine**].

Ownership

Values in Rust are bound to a single *owner*, which is the variable to which it is bound. When the owning variable goes out of scope, the value is freed. For example:

```
{
    let x = 69;
}
```

When `x` goes out of scope the value associated with it will also be released from memory. Because there can only be a single owner associated with a value, aliasing a variable is not allowed. This prevents any two parts of a Rust program from accessing a section of heap memory at the same time.

```
{
    let x = Thing::new();
    let y = x;
    println("()", x);
}
```

The above code does not compile because `x` has been moved to `y`, and there is an attempt to use `x` in the print statement. Using a variable as a function argument will also make it not available for use again. If a function uses some value that has been passing in as a argument it must return that same value if the scope outside that function uses it.

```
// value is comes from outside the scope of this function
fn foo(value: Bar) -> Bar {
    // do stuff with value, returning it and hence handing back ownership
    value
}
```

This is not practical, and gets even less practical once there are more arguments that must be passed back to the caller of the function. To this end Rust allows references to a value to be created. Creating a reference to a value in Rust is called borrowing. A reference to a value may be created and passed in to a function that uses it in some way, and that function does not need to thread the ownership of that variable back to the caller.

```
// instead of passing value1 and value2 in by value, create references to values
fn foo(value1: &Bar, value2: &Bar) -> u32 {
    // do stuff with value1 and value2 and return the answer
    123
}
```

There are two kinds of borrows: immutable and mutable. There may 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. The scope of borrow must always be shorter than that of the resource which it borrows.

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.

3.4 Representing Programs as Data Structures

3.5 Type checking

Chapter 4

Formal Definitions

This chapter attempts to formalize some of the concepts from Chapter **??**. The lambda calculus is introduced as a model of computation and as a system for reasoning about features found in programming languages. Types are introduced to the lambda calculus as well as other more complex extensions relevant to this project. The higher order polymorphic lambda calculus, known as System F ω, is included as well as a lifetime–calculus devised for this project.

4.1 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 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.

$\mathtt{t}$	$::=$	x	variables
	$\mid$	$\lambda x\,.\,\mathtt{t}$	function abstraction
	$\mid$	$\mathtt{t}\ \mathtt{t}$	function application

Figure 4.1: Grammer of the untyped lambda calculus.

4.1.1 Simple Types in the Lambda Calculus

The lambda calculus can be extended with the most simple of typing systems. In order to demonstrate how the lambda calculus can be used to reason about type systems, some additional constructs added. If Boolean literals some other concepts are added it becomes more clear how types are useful in programming languages.

Terms

t	::=	x	variables
	|	$\lambda x : T$. t	function abstraction, with type annotations
	|	$true$	true literal
	|	$false$	false literal
	|	if t then t else t	if-then-else expression
	|	t t	function application

Values

v	::=	$\lambda x : T$. t	lambda value
	|	$true$	true literal value
	|	$false$	false literal value

Types

T	::=	T $\rightarrow$ T	type of functions
	|	$Bool$	type of boolean literals

Figure 4.2: Lambda calculus extended with simple types and booleans.

Programs can be constructed using the language defined in Figure 4.1.1 that do not make any sense. For example:

baddef = if $\lambda x : Bool.x$ then $false$ else $true$

This program is constructed according to the grammar of terms. However, is not well type according to the typing rules in Figure **??**. Specifically, the type of the term found in the guard of the if-expression, $\lambda x : Bool.x$ has type $Bool \rightarrow Bool$ but has to be $Bool$ for the term to be considered well typed. It makes sense to restrict the kinds of values found in the guard of an if-then-else expression. Anything other than a boolean value in that place may indicate programmer error, which could be as simple as a typing mistake or a fundamental misunderstanding of what they are trying to express. Type systems help catch these kind of mistakes before program execution. Specifying these rules on top of the lambda calculus can be a very informative way of looking at programming language features. The rest of this chapter describes other extensions to the lambda calculus that model some of the programming concepts investigated in this report.

4.1.2 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 3.1.1. It is used to study implementations of polymorphism in programming languages.

As the simply typed lambda calculus allows for 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. This system can be used to reason about first order polymorphism, however more extensions to talk about higher order polymorphism. These extensions are introduced in Section 4.1.3.

Terms

t	::=	x	variables
	|	$\lambda x : T$. t	function abstraction, with type annotations
	|	t t	function application
	|	λX . t	type abstraction
	|	t[T]	type application

Values

v	::=	$\lambda x : T$. t	lambda value
	|	λX. t	type abstraction value

Types

T	::=	T $\rightarrow$ T	type of functions
	|	X	type variables
	|	$\forall X$. T	universal type

Figure 4.3: System F

Examples

System F models first order polymorphism in programming languages, such as aspect of the generics found in Java. Some examples of how System F can be used are given below:

```
id = λX.λx : X.x
boolId = id [Bool]
```

The example in Listing **??** 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 [Bool]`, which yields $\lambda x : Bool.x$, or the identify function over boolean values. Here X in the original definition of `id` has been instantiated with the type of Booleans.

4.1.3 System F ω

System F ω is form

Type Operators

4.2 Adding Lifetimes

showing how lifetimes are represented in this lanaguage

4.3 Final Typing Rules

Chapter 5

Requirements

*This section is *very* work in progress.*

5.1 Base Objectives

The base objectives for the project are outlined below.

- 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.

5.1.1 Design

5.1.2 Implementation

5.1.3 Tests

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

5.2 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 language 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.

5.3 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.

5.3.1 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.

5.3.2 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.

5.3.3 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.

5.3.4 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.

5.3.5 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.

5.3.6 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.

5.3.7 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.

5.3.8 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.

5.4 Acceptance Criteria and Testing

Should this section be related to KEY TESTS? Maybe this section should exist but should reference that chapter

The acceptance criteria of this program correspond to the functional requirements in Section 5.3. The finished project should pass the tests laid out in this section.

5.4.1 Parsing

Functional Requirement	5.3.1
Passing Criteria	The program should be able to parse valid source code and correctly report any errors that are encountered.
Tests	Test numbers

5.4.2 Type checking

Functional Requirement	5.3.2, 5.3.3, 5.3.4, 5.3.5
Passing Criteria	The type checker should detect any errors in the program.
Tests	Test numbers

5.4.3 Evaluating

Functional Requirement	5.3.6, 5.3.7, 5.3.8
Passing Criteria	Source code, provided by a file or through the interactive interpreter, should be type checked and evaluated.
Tests	Test numbers

Chapter 6

Implementation

This chapter describes how the ideas and type system described in Chapter 4 will be implemented. As stated in Chapter 5, one of the goals for this project is to build a program that will be able to parse, type check, and evaluate a language based on the lambda calculus. The most involved phase of implementation and the focus of this project involves the type checking phase, which includes lifetime checking of pointers and references, polymorphic type checking, and checking that the language of types themselves is well formed, or kind checking.

The requirements of this project influence the choice of the language of implementation. Haskell was chosen as it is well suited to the tasks of parsing, abstract syntax tree declaration, and is in general a practical language to work with.

6.1 The Haskell Language

Haskell is a general purpose, statically typed, functional language [**haskell**]. It has several desirable features for a language implementation language. These include:

- Algebraic data types, which are very good at representing abstract syntax trees, used to describe programming languages.
- Pattern matching, which is useful for deconstruct abstract syntax trees.
- A wealth of effective parsing libraries, over several paradigms.
- Idiomatic monadic programming, which can be used to reduce error handling and state threading boilerplate out of the logic of the program.

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. The language described in Chapter 4 has been implemented as a Haskell program. The rest of this chapter describes how this was accomplished in more specific detail.

6.2 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.

6.3 Terms

The Haskell representation of terms of the language are given in Listing 6.1. 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 6.1: Haskell representation of Terms.

6.4 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 6.2.

```
data Kind
    = KnStar
    | KnArr Kind Kind
    deriving (Eq, Show)
```

Listing 6.2: Haskell representation of Kinds.

6.5 Lifetime Checking

The syntax of lifetime is show in Listing 6.3.

explain what each of the lifetime consturctors mean

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

Listing 6.3: 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.

6.5.1 Ordering on Lifetimes

should probably write this, Ord instance for lifetimes.

6.6 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 6.4: Haskell representation of Types.

6.7 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 6.5: Partially applied error monad and langauge errors.

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

6.8 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 6.6. 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 6.6: 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 6.7: Envirnment and error monad stack.

6.9 Testing

Chapter 7

Key Tests

Chapter 8

Evaluation

8.1 Problems

Chapter 9

Conclusion

9.1 Further Work

Bibliography

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